An Efficient Locking Scheme for Path-based File Systems

Ritik Malhotra
University of California, Berkeley
ritik@berkeley.edu

ABSTRACT

Path-based file system drivers like FUSE, OSXFUSE, and CBFS have become increasingly popular tools for developers to create their own file system. Developing a file system using one of these drivers has many differences than developing one from scratch, including the fact that these drivers use paths to execute queries on files and folders instead of file handles or inode addresses. This limitation causes issues when trying to handle concurrent requests since these file systems cannot provide the same guarantees as non-path-based ones since they don't implement any locking mechanisms out of the box. To fix this problem, we propose an efficient and correct locking scheme to allow path-based file systems to properly handle concurrent requests.

Keywords

file systems, locking, FUSE, concurrency

1. INTRODUCTION

File system development is a difficult task for most developers, especially because of the dependency on writing file systems in the kernel-mode framework and having to deal with its various idiosyncrasies. There are various quirks and problems and a lot of boilerplate code that has to be written just to get a barebones working version of a file system up and running.

1.1 File System in Userspace

Fortunately, there have been various solutions to alleviate the difficulty of writing such file systems by different open source projects, commercial third-party extensions, and more. These solutions allow the developer to abstract the complexity of the kernel and write file systems without having to worry about some of the kernel's problems. One class of these solutions has emerged to be a popular choice for developers writing file systems in this day and age - FUSE, which stands for File System in Userspace [7]. FUSE, and its various incarnations - OSXFUSE for the Mac OS X operating system line [8] and Callback File System (CBFS) for the Windows operating system line [6] - allow a developer to write a file system completely in user-space, without having to touch the kernel or develop anything for it. For the sake of this paper, we refer to this class of tools as FUSE, since these solutions are all based off its original concept.

FUSE is able to accomplish this goal by installing its kernel extension and having that be the middleman communication library between the file system the developer develops in user-space, and the corresponding virtual file system (VFS) module that lives in kernel-mode and interacts with the various file system method calls made by the operating system. FUSE's kernel extension acts as a thin wrapper between the VFS and the developer's file system, simply acting as a unit that bounces calls from one end to the other, all while taking care of the kernel-mode quirks, edge cases, security, and more as a generic abstract solution for the developer to not have to think about.

On top of the simplicity FUSE provides, it has a myriad of other benefits for the developer to take advantage of, including:

- Access to multiple languages to write the file system instead of being limited to using C for a kernel-mode file system driver. The FUSE file system a developer gets to write is essentially treated as any other application running in user-space. This gives the developer the freedom to write it in any language, provided that it follows the contract set by the FUSE kernel extension and can communicate back and forth with it.
- Access to the network. Doing network requests in the kernel is not possible without some complicated machinery, but is the complete opposite with FUSE file systems. Since

they run like other applications, they also have the freedom to pick up any regular library and use it in the file system, including ones that handle network requests. This allows for easier development of network file systems and network-connected distributed file systems, which are becoming more popular recently.

However, FUSE is not without its limitations and has a few drawbacks versus writing a file system purely from scratch [4], including:

- Much slower performance due to the heavy context switching required between user-space and kernel-mode. This is expected since each call that comes to the FUSE kernel extension has to be transported across the user-kernel boundary to be queried on the user-space file system before returning the appropriate data to the caller (which is another round trip through kernel-mode and back up the calling application in user-space).
- Unable to get granular control over memory and cache management. Since FUSE file systems don't live in the kernel, we're restricted on the amount of low-level control we have over things like memory management, data caching, and other kernel-specific functions.
- Queries are done repeatedly via paths instead of an ID-based or pointer-based system. This is problematic since paths aren't the best form of identifying a node in a file system, as we later discuss in this paper.

The last issue presented, related to path-based queries, is the one we focus on in this paper. We delve into this issue to determine the ramifications of such a system, what limitations it poses, what potential solutions look like, how we solve the problem, and the benefits from doing so.

1.2 Path-based Queries

One of the biggest differences with a FUSE file system and a regular file system is the fact that method queries are done via paths to file system objects, rather than an inode identifier, a file handle, or any other sort of pointer-based query method. This design choice is done within FUSE's kernel extension, and is done by choice, not by some limitation of the underlying kernel software. FUSE's mantra resides with keeping the kernel extension thin in order to keep performance as high as possible and act purely as a pass-through layer. This in turn means that FUSE doesn't keep a mapping between the path to an object and its inode ID, but rather passes the path that it receives from the VFS module up to the user-space file system implementation. From there on, the user-space file system is expected to keep its own consistent mapping of paths to the appropriate data structures containing the information about the object at that path.

This makes sense purely on a development side - let the developers handle their own data structures and interfere as little as possible. This is generally a sign of a good API that abstracts out the right things, and standalone sounds like a harmless feature. Unfortunately this concept introduces more problems when combined with other aspects of the file system, which we introduce later in this paper.

1.3 Concurrency

Like most regular file systems, FUSE file systems are inherently designed to support concurrent operations on it. The FUSE kernel extension handles concurrent requests and passes them in a non-blocking fashion up to the user-space file system implementation and leaves it to handle proper concurrency control, serialization, etc. This is fantastic for performance reasons, but pushes the burden of handling parallelism within the file system on to the developer.

Again, standalone, this is a great feature for FUSE file systems to support since it helps with performance, but combined with other quirks of how FUSE works, this poses a problem.

1.4 Concurrency in a Path-based File System

When we combine both these concepts - concurrency and path-based querying - there's a larger issue that pops up: how do we properly serialize requests that could potentially run into conflicts or concurrent modification exceptions on the file system's core data structures.

To depict this problem, let's take a look at an example:

Algorithm 1 Example of Problem

```
1: procedure InterleavingThreadsExample
2: Thread 1:
3:     $write('/a/b', DataChunk1)$
4: Thread 2:
5:     $rename('/a', '/c')$
6: Thread 1:
7:     $write('/a/b', DataChunk2)$
```

This example shows two interleaving threads (presumably two different applications working on the file system at the same time) acting on conflicting data. Let's take a look at what really happened here: Thread #1 is undergoing a series of write() requests to write data to the file at /a/b. It executes the write requests in chunks defined by the kernel, so it has to make multiple write() calls to get all the data written to the file. Thread #2 is unaware of thread #1's intention to write data to a file that is a child node of the folder thread #2 is modifying. Thread #2 continues its operation and renames the folder at /a to /c. When thread #1 makes another write() call to write the second chunk of data, the file system is in an inconsistent state from what thread #1 thought it was at after writing the first chunk of data. The file at /a/b no longer exists and is now located at /c/b. But since thread #1 has no way of knowing that, it receives an -ENOENT error (file not found) and the write fails. Moreover, if the user-space file system doesn't have transaction-based writes implemented, the file at /c/b (originally at /a/b) now has incomplete data since the write transaction did not finish.

This problem is a critical issue for any FUSE file system, and only stems from the fact that the file system is queried by paths to the object. If the file system was queried by an ID or a pointer, this problem wouldn't arise because the method calls would occur behind a level of indirection (the ID or pointer), which wouldn't have to deal with the underlying path being modified during a transaction. But in a path-based world, because the parent folder of the object being written to was modified in a separate thread, the path in the second write operation was no longer valid. On top of this inconsistency, there's no way for the FUSE file system to communicate this underlying change to the application in thread #1. Nor can thread #1 arbitrarily decide to "correct" the path because there's no reasonable way for the file system to know what the proper behavior is - does the application really want to write to the new location of the file, or does it actually want to go forward with a write operation on the old path?

2. POTENTIAL APPROACHES

To tackle this problem, we first consider a few approaches that are borrowed from other system designs or are trivial to think of and consider. The following section considers three different approaches, all of which are not suitable for this problem, but are good primers to understand the problem and borrow from when constructing the actual solution.

2.1 Global Locking

The simplest way to tackle the problem of concurrency in a path-based query system is to disable concurrency. By doing so, we effectively turn the file system into a single-threaded system, which effectively mitigates the problem at hand. No longer does the file system have to deal with interleaving threads modifying related data and leaving the file system in an inconsistent and corrupt state. If the file system has a global access lock for all operations, it immediately serializes all file system method calls and turns the file system into a single-threaded system.

But even this approach is limited as it still puts the burden on the developer to ensure that certain file system operations are transactional, like reads and writes. Otherwise, the file system is still prone to the same error described in the example above, even if the requests are serialized.

On top of that, adding a global lock on the entire file system also drastically hurts performance and would render the file system useless for certain applications that rely on concurrency. A performance hit this drastic is unacceptable, so we have to look for a better way to handle this situation.

2.2 Linux File System Locking Scheme

One approach to consider would be to use the locking scheme that the Linux kernel exposes for its file systems. The Linux kernel's file system locking specifications are special in that they implement what's called advisory locking, which is a locking scheme that is entirely optional for applications to use and check for, but is exposed for applications to use if they wish to do so. Underneath the advisory locking scheme are rules that govern how the file system handles acquiring and releasing locks based on the operation being conducted, which, as expected, is designed for a pointer-based query file system and not a path-based query file system.

Unfortunately, this approach has two problems with it when applied to a FUSE file system:

1. Advisory locking is unsuitable for our file system due to its optional enforcement nature. We need a solution that will provide guaranteed lock enforcement; otherwise the file system can go into an inconsistent state. Linux file systems don't run the same risk of inconsistency without locks because they're inherently a pointer-based query system and won't run into the issue of stale paths like FUSE file systems will.

2. The locking rules don't match the requirements for what a path-based file system needs. The trivial example presented earlier in this paper fails the same way even when considering the Linux file system locking rules.

These two limitations render this approach a dead end; however, this approach is a great starting point to come up with a locking scheme for FUSE file systems.

2.3 Snapshotting

Another approach is to use snapshots for every single operation as a way of marking the state of the file system at the start of every operation. This allows operations to work concurrently on the same file system data structures, but at different states. This way, the operations are guaranteed to have a consistent state of the file system at any given point during their execution since they get full isolation from other operations and are guaranteed that no other operation will intrude on their snapshot.

While this makes sense, this approach has one flaw: merging the snapshot's changes back into the master file system. The same challenge applies when trying to merge the changes back into the master state, which would need some sort of serialization to ensure concurrent merges don't conflict, or have a conflict resolution algorithm to solve those conflicts. This approach eventually devolves back into having a single global lock, so we must look elsewhere and think of a better design.

3. SYSTEM DESIGN

We know that the main problem of locking in a FUSE file system is the fact that as paths compound on one another and rack up more and more ancestors in the path, there are more areas for inconsistencies to occur. To get around this issue and ensure proper serialization we must design a locking system to acquire locks for all the ancestors in a path that's being operated on, so that none of the ancestors of the path being operated on can be changed from underneath in a separate thread while the operation is happening.

3.1 Types of Locks

There are two types of operations that a file system can perform on its objects: reads and writes on the object's data or metadata. The logical step for our locking scheme would be to use a read/write lock for each path. On top of that, we also propose having two different kinds of locks in our scheme:

1. Path locks - These locks are responsible for locking any operation being conducted on the path where an object lives. This even applies for operations as simple as accessing a path.
2. Data locks - These locks are responsible for locking any operation being conducted on the actual data or metadata of an object. To acquire a data lock on a path, a path-read lock must have been acquired on the path first.

This gives us a total of four locks - path-read, path-write, data-read, and data-write - to work with.

Having two different types of locks is necessary to get more granular isolation of operations. Operations that affect the path don't affect the data or metadata of an object, and the opposite holds true as well. This allows operations that are simply accessing the path of an object and another operation modifying the data of that same path to happen simultaneously, which is not the norm in most other file systems. This granularity still results in correct serialization of dependent operations, while providing better parallelism and performance than just having generic read/write locks without this granularity.

3.2 Ancestor Locking

Ancestor locking is the main aspect of this locking scheme that makes it different from other file systems. As mentioned before, to ensure that paths remain consistent across file system method calls, we need to impose locks on all ancestors of a path being operated on [2]. For example, if an operation needs to be conducted on a path /a/b/c/d/e, the operation will need to acquire path-read locks on /, /a, /a/b, /a/b/c, and /a/b/c/d (and some sort of lock on /a/b/c/d/e, depending on the operation) since each of those paths is being "accessed" to ensure that it remains consistent throughout the duration of the operation.

We can illustrate how this ancestor locking mechanism prevents concurrency issues. Let's say a thread is trying to rename /a/b/c to /a/b/d while another thread is trying to create /a/b/c/e. The first thread acquires path-read locks on / and /a and path-write locks on /a/b/c and /a/b/d. The second thread tries to acquire path-read locks on /, /a, and /a/b and data-write locks on /a/b/c and /a/b/c/e. But the data-write locks the second thread wants require path-read locks as a prerequisite, which are then blocked by the first thread's path-write lock on /a/b/c. These two operations will be serialized properly because they're trying to obtain a conflicting lock on /a/b/c.

3.3 Locking Order

The next thing to consider is the ordering of lock acquisition. Without a proper ordering scheme, we have no guarantee that our locking scheme won't result in a deadlock. For example, if we ever have two threads that are trying to acquire two identical locks as one another, but execute the lock acquisition in different orders, the file system will run into a deadlock. With a consistent lock acquisition ordering rule, we can guarantee that two threads will never deadlock trying to acquire locks. This is because it will be impossible for a thread to acquire a lock in a different order than another thread, which means that if a thread is waiting to acquire a lock, the thread that has the lock will not need to lock anything that the waiting thread has already locked (otherwise the lock ordering rule would have been violated).

To ensure consistent ordering, we propose acquiring locks in ancestor-first order. So for example, if we're trying to acquire a lock on path /a/b/c, we'll acquire a lock on /, then /a, then /a/b, and then finally /a/b/c. Ordering on the same level (i.e. choosing between /a/b/c and /a/b/d) will be done in lexicographic order. When unlocking, we follow the exact opposite order of how locks were acquired.

For example, if an operation is trying to acquire a lock on /a/b/c, /a/b/d, and /a/e/f, the operation would acquire the locks in the following order: /, /a, /a/b, /a/e, /a/b/c, /a/b/d, /a/e/f. Unlocking would be done in reverse order.

This is the only way we can order our locks without having to use any other knowledge about the file system other than what FUSE sends up to the user-space implementation (which is the path).

3.4 Locking Scheme

With a solid foundation set up of how locks will be managed, acquired, and released, we can now define a clear scheme on how to actually do the locking mechanics. For our purposes, all file system operations fall into five classes:

1. Creating or deleting an object
2. Accessing an object's data or metadata
3. Modifying an object's data or metadata
4. Renaming an object
5. Moving an object (cross-directory rename)

We describe the locking scheme for each of these classes below and depict the locking scenario with an example.

3.4.1 Creating or deleting an object

Creating or deleting an object is an interesting case in that the operation needs to "modify" itself and ensure that its parent is not modified during the process. Modifying itself is self-explanatory - it needs to either write or remove data from its own path. Keeping its parent intact is necessary to ensure that the parent doesn't get deleted or renamed mid-way, resulting in an inconsistent state. To achieve this, we require a data-write and path-write lock on the path being created or deleted, a path-read lock on the parent, and a path-read lock on all the other ancestors. The path-read lock on the parent ensures that it won't be modified during the operation, but is still open to other operations being conducted inside it (as long as they don't violate the other locks acquired).

The FUSE file system methods that fall under this class are mknod(), mkdir(), unlink(), and rmdir().

Let's take a look at an example:

Algorithm 2 Delete Locking Example

```
 1: procedure UNLINK(/a/b/c.txt)
 2:     lock(/).path().read()
 3:     lock(/a).path().read()
 4:     lock(/a/b).path().read()
 5:     lock(/a/b).data().read()
 6:     lock(/a/b/c.txt).path().write()
 7:     lock(/a/b/c.txt).data().write()
 8:     Execute unlink() operations
 9:     unlock(/a/b/c.txt).data().write()
10:     unlock(/a/b/c.txt).path().write()
11:     unlock(/a/b).data().read()
12:     unlock(/a/b).path().read()
13:     unlock(/a).path().read()
14:     unlock(/).path().read()
```

3.4.2 Accessing an object's data or metadata

Accessing data or metadata on an object is a relatively trivial case. It requires only data-read permission on the object itself and nothing more since it's a non-destructive function. This means that a data-read lock is required on the path being accessed and a path-read lock on all its ancestors.

The FUSE file system methods that fall under this class are getattr(), getxattr(), listxattr(), statfs(), read(), and readdir().

Let's take a look at an example:

Algorithm 3 Get-Attributes Locking Example

```
 1: procedure GETATTR(/a/b/c.txt)
 2:     lock(/).path().read()
 3:     lock(/a).path().read()
 4:     lock(/a/b).path().read()
 5:     lock(/a/b/c.txt).path().read()
 6:     lock(/a/b/c.txt).data().read()
 7:     Execute getattr() operations
 8:     unlock(/a/b/c.txt).data().read()
 9:     unlock(/a/b/c.txt).path().read()
10:     unlock(/a/b).path().read()
11:     unlock(/a).path().read()
12:     unlock(/).path().read()
```

3.4.3 Modifying an object's data or metadata

Modifying data or metadata on an object is very similar to accessing data or metadata on it. Instead of the read permission required for accessing data, this case requires data-write permissions instead. This results in a data-write lock on the path being modified and a path-read lock on all its ancestors.

The FUSE file system methods that fall under this class are chmod(), chown(), setxattr(), removexattr(), utimens(), truncate(), and write().

Let's take a look at an example:

Algorithm 4 Truncate Locking Example

```
 1: procedure TRUNCATE(/a/b/c.txt, 4096)
 2:     lock(/).path().read()
 3:     lock(/a).path().read()
 4:     lock(/a/b).path().read()
 5:     lock(/a/b).data().read()
 6:     lock(/a/b/c.txt).path().read()
 7:     lock(/a/b/c.txt).data().write()
 8:     Execute truncate() operations
 9:     unlock(/a/b/c.txt).data().write()
10:     unlock(/a/b/c.txt).path().read()
11:     unlock(/a/b).data().read()
12:     unlock(/a/b).path().read()
13:     unlock(/a).path().read()
14:     unlock(/).path().read()
```

3.4.4 *Renaming an object*

Renaming an object is effectively the equivalent of modifying an object's path. We draw a distinction between renames and moves by drawing the line at in-directory renames and cross-directory renames. A rename operation is defined as purely a name change of an object (an in-directory rename), whereas a move operation is defined as a parent folder change of an object and, optionally, a name change (a cross-directory rename).

This operation requires touching three things: the path being renamed, the new path being renamed to, and the parent folder of the path. Since this operation only deals with paths, we need a path-write lock on the path being renamed, the path being renamed to, and the parent folder of both of these paths, and a path-read lock on all the other ancestors.

The FUSE file system method for this class is rename(), but only in the case of an in-directory rename.

Let's take a look at an example:

Algorithm 5 Rename Locking Example

```
 1: procedure RENAME(/a/b/c.txt, d.txt)
 2:     lock(/).path().read()
 3:     lock(/a).path().read()
 4:     lock(/a/b).path().read()
 5:     lock(/a/b).data().read()
 6:     lock(/a/b/c.txt).path().write()
 7:     lock(/a/b/c.txt).data().write()
 8:     lock(/a/b/d.txt).path().write()
 9:     lock(/a/b/d.txt).data().write()
10:     Execute rename() operations
11:     unlock(/a/b/d.txt).data().write()
12:     unlock(/a/b/d.txt).path().write()
13:     unlock(/a/b/c.txt).data().write()
14:     unlock(/a/b/c.txt).path().write()
15:     unlock(/a/b).data().read()
16:     unlock(/a/b).path().read()
17:     unlock(/a).path().read()
18:     unlock(/).path().read()
```

3.4.5 *Moving an object (cross-directory rename)*

Moving an object is the most complicated class that we have to consider. Moving an object touches four different things: the original path, the original path's parent, the new path, and the new path's parent. Each of these needs a path-write lock, and all their other ancestors require a path-read lock.

On top of this, these operations also require various error checks to be intertwined within the locking scheme to ensure correctness. We have to check for the following two error cases to ensure no circular renames are happening:

1. If the source object's parent folder is equal to the destination object or is a descendant of the destination object, return -ENOTEMPTY.
2. If the destination object's parent folder is equal to the source object or is a descendant of the source object, return -ELOOP.

These error checks need to be made after the appropriate locks have been acquired to ensure there aren't any race conditions with other competing threads. And since the locks are implemented as reentrant locks, if one of these error conditions does exist and we do have a circular rename, the file system won't go into a deadlock state since it'll be the same thread trying to acquire a lock it already has (which will be allowed thanks to the reentrant property of the locks we're using).

The FUSE file system method for this class is also rename(), but only in the case of a cross-directory rename.

Let's take a look at an example:

Algorithm 6 Move Locking Example

```
 1: procedure RENAME(/a/b/c, /a/d/e)
 2:     lock(/).path().read()
 3:     lock(/a).path().read()
 4:     lock(/a/b).path().read()
 5:     lock(/a/b).data().read()
 6:     lock(/a/d).path().read()
 7:     lock(/a/d).data().read()
 8:     lock(/a/b/c).path().write()
 9:     lock(/a/b/c).data().write()
10:     lock(/a/d/e).path().write()
11:     lock(/a/d/e).data().write()
12:     if object(/a/b) = object(/a/d/e) then
13:         return -ENOTEMPTY
14:     if object(/a/b) = desc(/a/d/e) then
15:         return -ENOTEMPTY
16:     if object(/a/d) = object(/a/b/c) then
17:         return -ELOOP
18:     if object(/a/d) = desc(/a/b/c) then
19:         return -ELOOP
20:     Execute rename() operations
21:     unlock(/a/d/e).data().write()
22:     unlock(/a/d/e).path().write()
23:     unlock(/a/b/c).data().write()
24:     unlock(/a/b/c).path().write()
25:     unlock(/a/d).data().read()
26:     unlock(/a/d).path().read()
27:     unlock(/a/b).data().read()
28:     unlock(/a/b).path().read()
29:     unlock(/a).path().read()
30:     unlock(/).path().read()
```

4. EVALUATION

The evaluation of this locking scheme is split up into two parts: correctness and performance. Correctness is discussed with proofs to show that the locking scheme works correctly in all situations. Performance is displayed by analyzing the system under heavy concurrent operation loads. Due to the nature of this problem, the performance isn't as important as the correctness of the scheme is, since without this locking mechanism FUSE file systems are simply prone to errors.

4.1 Correctness

The most important part of this locking scheme is to prove correctness. We have already shown in the above examples that this locking scheme is correct when applying locks for each operation. In this section, we prove that this scheme is technically correct if it doesn't run into any deadlocks. To do this, we apply a modified version of the System V Linux file system locking proof [9] to our system here.

First, we define any moment in which we have a partial ordering of the objects as $A < B$ *iff A is an ancestor of B*. We assume that that ordering can change, but the following rules must hold true:

1. If a non-move operation holds a lock on A and attempts to acquire a lock on B, A will remain the parent of B until we can acquire the lock on B. This is because only a move operation can change the parent of an object and for it to do so, it would need to lock the parent, in this case A (which is locked).

2. If a move operation holds all its locks, the ordering of objects in the file system will not change. This is because all other renames and moves that would potentially change the ordering for the objects that the move operation cares about will be blocked by the path-read locks held on all the ancestors.

3. Locks are always acquired in an ancestor-first, lexicographic order, so two operations acquiring conflicting locks can never deadlock with one another.

Given these rules, let's consider a trivial deadlock case in which all operations are blocked on acquiring some lock, but already hold at least one lock. By rule #3, any operation holding a lock can only be waiting for a lock that is lower on the ordering list. Therefore, the operation holding the "largest" lock (the lock that would be first on the ordering list) can always make forward progress making this deadlock not possible.

Furthermore, any contended lock has to be held by an operation that has a child lock that is also contended. Suppose that the lock is held by an operation that is not a move operation. Then the lock this operation is blocked on belongs to a child of that object because of rule #1. This means that one of the blocking operations is a move operation because otherwise, the set of contended locks would be infinite. Each of them would have a contended child, which is impossible since we assume there are no loops and no object is its own descendant.

Lastly, consider a contended lock blocking a move operation. One of its descendants has to be locked by the move operation (otherwise we hit the infinite set of contended locks case again). This doesn't make sense because for this situation to happen, the move operation would have had to acquire locks out of order. Rule #2 says that the order of objects in the file system can't have changed and rule #3 says that ancestors are locked first. This is a contradiction, so a deadlock is impossible.

4.2 Performance

To further evaluate the system, we also do a quantitative analysis of the impact the locking scheme has on the file system. Before jumping into the numbers, it's important to note two important facets about performance and how it relates to other file systems, which we discuss in the following sections.

4.2.1 *Performance is not a critical metric*

Performance is not a critical aspect of this discussion. Contrary to other papers, performance is not a key metric of success for this locking scheme. With the current state of affairs of FUSE file systems, without a proper locking scheme, FUSE file systems are simply incorrect and will not operate as expected against concurrent operations. Therefore, the benchmark to compare any quantitative performance metrics is against a broken system, which doesn't make sense for this discussion. Instead, we compare it to a FUSE file system which has concurrency disabled (i.e. single-threaded mode), which isn't as interesting, but at least gives us an indication of what to expect from before and after.

4.2.2 *Our locking scheme is not relatable to other file systems' locking schemes*

The locking system that we're measuring in this paper is **not** related to other file system's locking systems. Other file systems that implement locking schemes do so for *external locking*. External locking in this case refers to exposing an external API for developers of applications to call to lock file system objects. For example, Linux file systems follow an advisory locking specification that applications can use to lock objects in the file system, but is not required.

This is not the same as the locking system described in this paper. Our locking system is designed as an *internal locking* mechanism that protects internal file system data structures from dealing with concurrent modification exceptions and for keeping state consistent. The easiest way to think about this is if we removed the external locking components of other file systems, they would still operate correctly. But if we removed our internal locking components from a FUSE file system, the file system would no longer work correctly when bombarded with concurrent operations.

4.2.3 *Analysis environment*

With that said, we can still take a look at a few key metrics to determine the efficacy of our system when compared to the baseline FUSE file system running in single-threaded mode. We conduct all our quantitative analysis and collect all metrics on a 15" Macbook Pro Retina (2012 model) with a quad-core 2.3 GHz Intel Core i7 processor, 16GB 1600 MHz DDR3 RAM, and a 256 GB Apple SSD. A sample FUSE file system was created, running purely in memory. The file system contained 256 files and folders, with the folder tree going up to four levels deep. A sample test suite was also created favoring a heavy write workload, consisting of 50% write operations and 50% read operations.

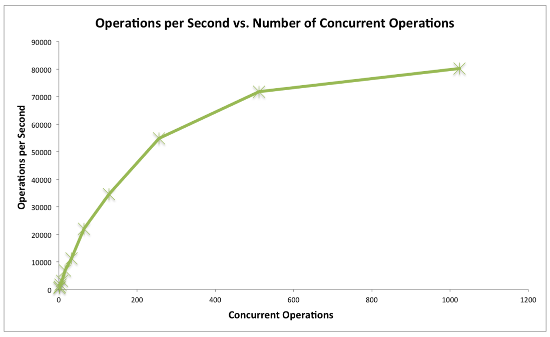

Figure 1: Operations per Second

The operations were a mix of all file system methods.

4.2.4 *Analysis of operations per second*

The first interesting metric is taking a look at the operations per second the file system can handle. In a single-threaded file system, we observe that the number of operations per second stays constant as we add more concurrent operations, but in the case of a multi-threaded file system with our locking scheme, we see much better results, as seen in Figure 1 above.

This logarithmic curve is expected since the growth of operations per second starts to slow down as we get more concurrent operations. The reasoning behind this is due to the increased amount of lock contention as we get higher concurrency, which is a concept explored in the next section.

4.2.5 *Analysis of lock contention*

The next metric to look to is the amount of times a lock is contended due to a concurrent operation. Figure 2 below depicts the analysis and shows that lock contention grows linearly as the number of concurrent operations increases. Again, this is what we expect since the higher number of concurrent operations we have, the more the locks will be held and be contended for.

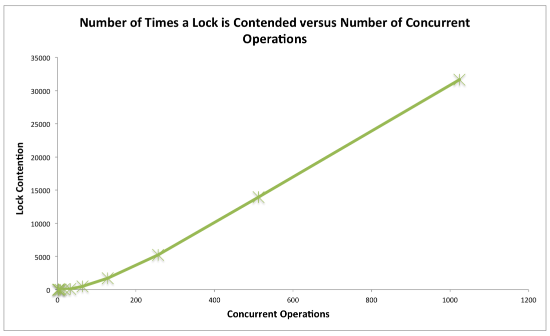

Figure 2: Number of Times a Lock is Contended

5. FUTURE RESEARCH

The locking approach outlined in this paper is a good start to allowing concurrent operations in a FUSE file system; however, improvements can always be made. A few potential avenues of future research include:

- More granular locking for each type of operation. Our implementation splits up the locking structure into two locks - path and data locks - which provide a higher level of granularity than just having a single lock for each path. There may be more scope in looking into if we can split the locks even more depending on what data is dependent on one another. For example, when writing data to a file, the write() operation is reliant on having access to the data blocks and access to update the file's size metadata field (and others, depending on the implementation of the write() function), but may not need access to certain other metadata fields. In this example, we could potentially granularize the locks such that writes don't have to lock up all metadata fields, so other operations can read and modify metadata fields that aren't related to write().
- Implementing the locking scheme directly in the FUSE kernel extension instead of using it as a library in the user-space file system implementation. This would improve performance since all locking (and some error handling) would be handled in the kernel rather than in user-space, which would allow the kernel extension to better queue up and send requests to the user-space implementation.
- Implement the concept of an inode in the FUSE kernel extension and send up inode IDs rather than paths. This would eliminate the locking problem entirely since FUSE file systems would operate like regular file systems and we'd be able to take advantage of existing locking solutions. This would require the most amount of investigation and work, but is the best avenue for future research.

6. CONCLUSION

The existing state of FUSE file systems is not at the level we'd want for production use. Without a proper locking system to handle concurrent requests, these file systems are unreliable and present a problem that may not be immediately obvious to the developer. The few existing FUSE file systems that do implement their own locking systems are forced to make their own rules, sink expensive developer time into doing so, and don't follow a strict standard. This is problematic for a feature that's expected to exist right out of the box.

We propose a solution to this issue and present a correct and efficient internal path-based locking scheme for all FUSE file systems to follow. Our solution provides granular locking that allows for more concurrency than a simple system. We also prove its correctness and show the performance increases with our locking system versus without it.

Acknowledgement

We wish to thank John Kubiatowicz for helpful feedback on an earlier poster of this work and Brian Miller for his help and guidance around file systems theory.

References

[1] Frank Schmuck and Roger Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters". In: *USENIX Proceedings of the Conference on File and Storage Technologies* (2002), pp. 231–244. URL: http://www.cse.buffalo.edu/faculty/tkosar/cse710_spring14/papers/gpfs.pdf.

[2] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: *Association for Computer Machinery Symposium on Operating Systems Principles* (2003). URL: http://static.googleusercontent.com/media/research.google.com/en/us/archive/gfs-sosp2003.pdf.

[3] Sage A. Weil et al. "Ceph: A Scalable, High-Performance Distributed File System". In: (2006). URL: http://www.ssrc.ucsc.edu/Papers/weil-osdi06.pdf.

[4] Sage A. Weil. *Linus vs. FUSE.* 2011. URL: http://ceph.com/dev-notes/linus-vs-fuse/.

[5] Paul Krzyzanowski. *Distributed File Systems Design.* 2012. URL: https://www.cs.rutgers.edu/~pxk/417/notes/15-nas.html.

[6] *Callback File System.* URL: https://www.eldos.com/cbfs/.

[7] *Filesystem in Userspace.* URL: http://fuse.sourceforge.net/.

[8] *FUSE for OS X: The easiest and fastest way to create file systems for OS X.* URL: https://osxfuse.github.io/.

[9] *Linux File Systems Documentation: Directory Locking.* URL: https://www.kernel.org/doc/Documentation/filesystems/directory-locking.

[10] O. Rodeh and A. Teperman. "zFS - A Scalable distributed File System using Object Disks". In: (). URL: http://ibm.co/138oGvt.