Finesse: Flow of the open request

Tony Mason

November 11, 2019

1 Overview

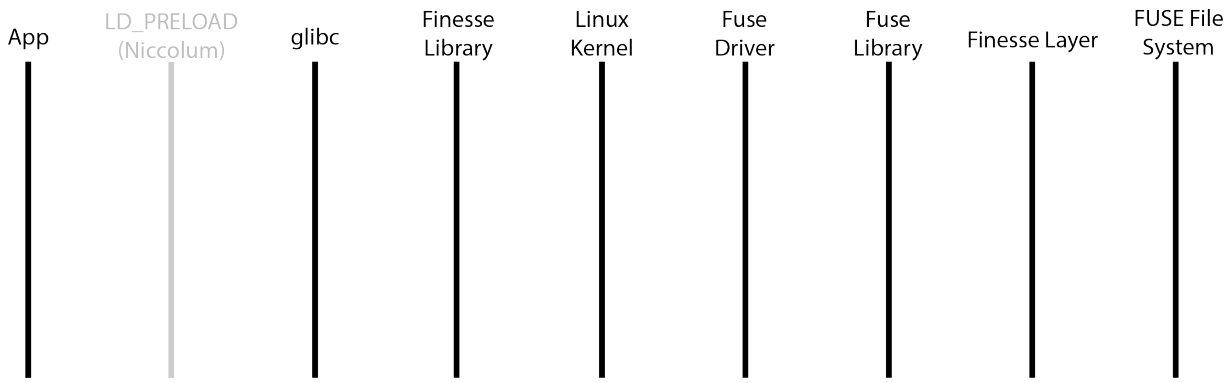
The Finesse project attempts to add a message passing layer between applications and a FUSE file system that has the integrated support for Finesse. This is done in a way that attempts to be as transparent as possible to the application and the FUSE file system.

Thus, the key goals are:

- Application compatibility — an application should run, without modification.
- Application extensibility — an application should be able to directly use the Finesse APIs directly, including those that mirror existing POSIX API calls as well as enhanced/augmented calls.
- Support for LD_PRELOAD — by providing an LD_PRELOAD interposition layer ("niccolum") standard API calls may be substituted with calls to the Finesse layer. This is entirely transparent to the application.
- Support for mixed legacy and Finesse access — calls can be sent via POSIX system calls or through the Finesse layer and work, regardless of the mixing of the paths.
- Finesse uses a message passing model for communications with the FUSE file system

2 Finesse Open Flow

In this section I describe the flow of messages from an application to the finesse layer via a series of flow diagrams showing the intended exchange of calls and messages; note that this is conceptual and may omit steps or vary from the actual implementation.



Base Flow Diagram

App this is the application that is using the FUSE file system.

LD_PRELOAD this is the interposition call; thus, instead of calling `open` in *glibc*, this call is intercepted by the preload handler. I describe this in greater detail in §3.

glibc this is the default C standard library implementation on the underlying system. It does not actually require that it be the GNU implementation of Lib C, but that is the most common implementation in use on systems where we will be testing.

Finesse Library this is the application side logic for integrating the handle-driven logic of the standard POSIX file system API and Finesse. When Finesse receives a request, it uses the handle (file descriptor) to determine if it has state for handling the request. For requests without handles (e.g., open), it will send a *lookup* message to the Finesse FUSE library layer. If successful, this will return a handle (UUID) for subsequent accesses to this file: note that the UUID is intended to be unique to the file and on the Finesse FUSE library side it is generally associated with the *inode* number of the file. Finesse may interact with *glibc* in some situations where there is kernel maintained state that must be properly updated (e.g., open, read, and write).

Linux kernel this is the layer at which *system calls* are handled. The expectation is that *glibc* will ultimately invoke one of these system calls.

FUSE Driver this is the kernel mode driver that translates VFS calls into messages to the FUSE library.

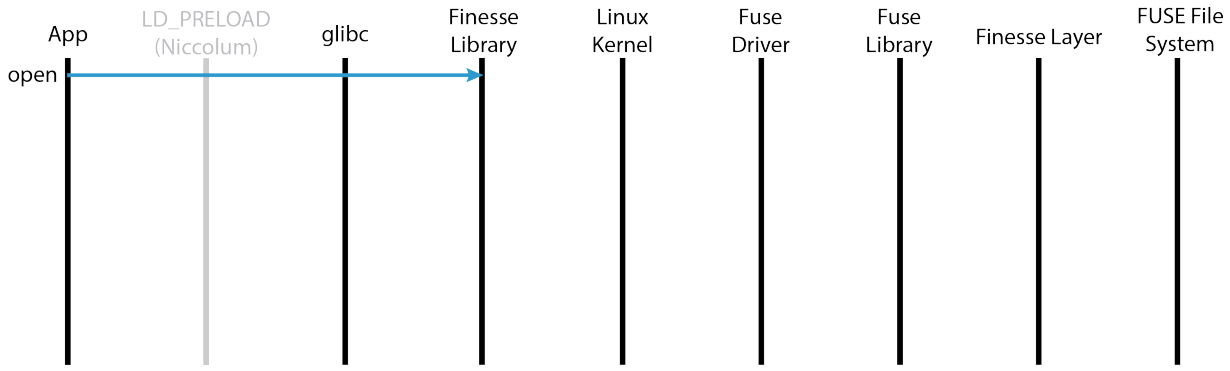
FUSE Library this is the user mode FUSE library that translates messages from the FUSE Kernel driver into calls to the FUSE File system.

Finesse Layer this is the Finesse FUSE library side support for Finesse messages (and hence Finesse functionality) in terms of the calls to the FUSE file system. Note that currently this only works with the low-level FUSE API. This should not be problematic, because the high-level FUSE API is now implemented in terms of the low-elvel FUSE API. The low-level FUSE API is an asynchronous request (event) driven model.

FUSE File system this is the user mode file system implementation; we expect that it will normally be written to the FUSE library API, but may also (optionally) implement Finesse specific functionality (if/when we decide to introduce such additional functionality).

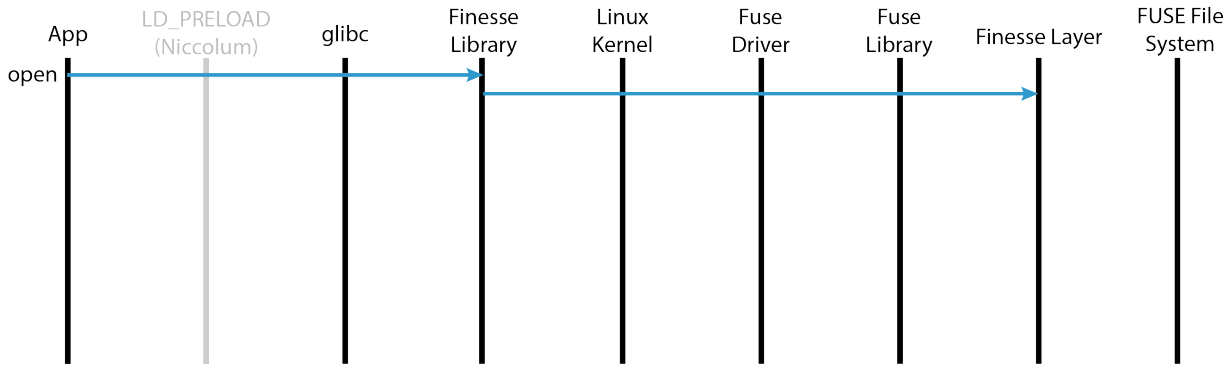
When an application issues an `open` call, either directly (as shown below) or via the LD_PRELOAD interface (which is showed in grey), the call is primarily handled by the Finesse library. Finesse will

only handle calls that are directed to a known Finesse enabled FUSE file system; otherwise, this call should redirect the call to *glibc* or rejected with an error condition.



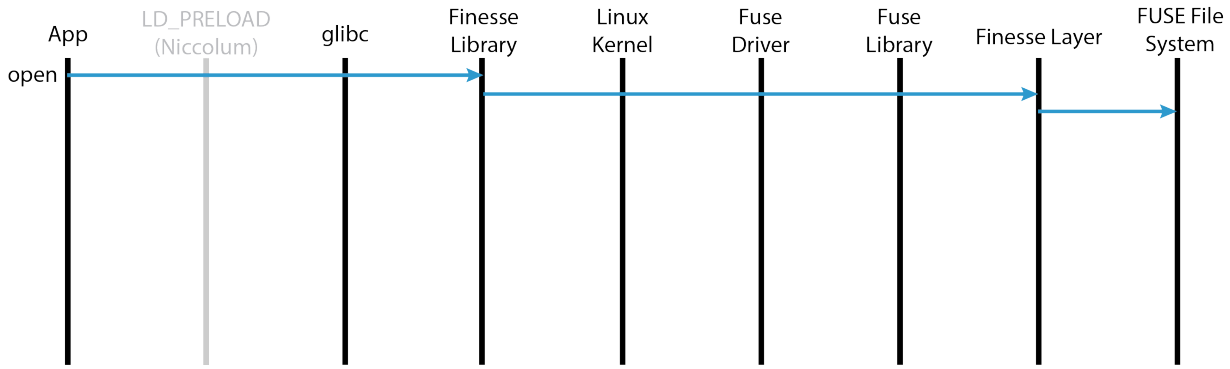
Open request from application

Assuming the call is to a Finesse aware file system, the Finesse library will send a *message* to the Finesse layer in the FUSE library that attempts to convert the provided name into a Finesse handle (UUID) that can be subsequently used for other operations (including *release*). In the current implementation, the UUID is in a one-to-one mapping with the inode number.



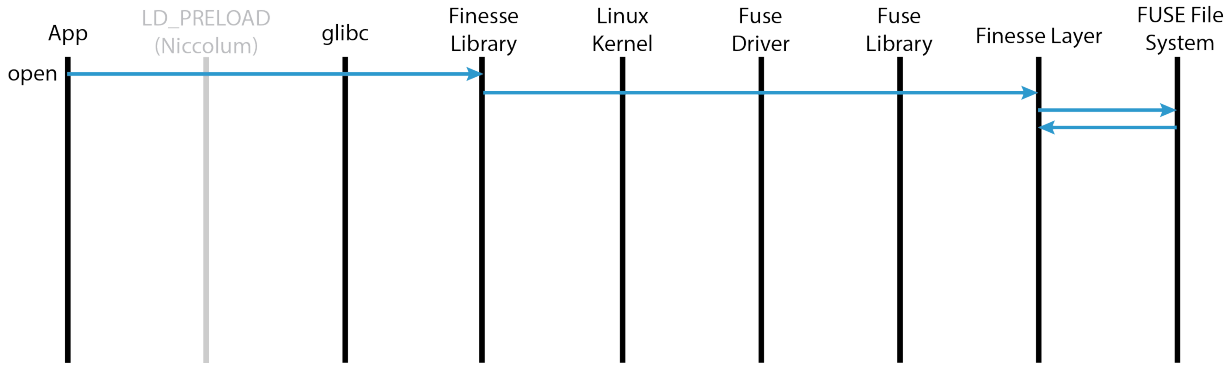
Finesse Library sends *lookup* request to FUSE/Finesse Layer.

The Finesse Layer inside the FUSE library obtains the inode number (or a failure) using the FUSE file system, via the *lookup* call. The Finesse side can then associate the UUID with the inode number within its own data structures, which will facilitate future operations; Finesse is using the UUID while the FUSE file system is using an inode number. Finesse may store additional state beyond the UUID/inode mapping as well.



FUSE/Finesse Layer invokes the FUSE file system.

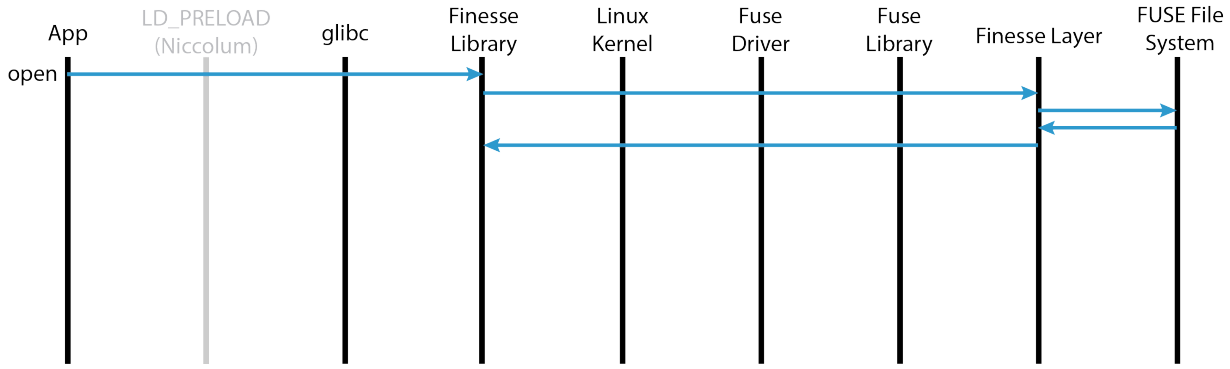
The FUSE file system will either return a valid inode number or an error.



FUSE file system returns error or valid inode number.

The Finesse layer will then do any "bookkeeping" necessary, such as finding an existing UUID for that inode number, or creating a new UUID if this inode number is not present in its own tracking structures.

It can then send an appropriate response message back to the Finesse Library.

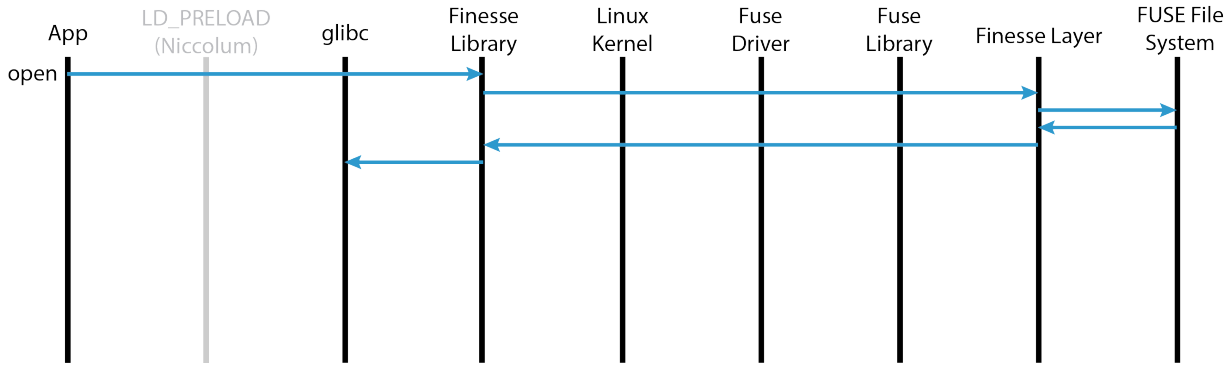


FUSE/Finesse Layer sends response message back to Finesse Library.

Depending upon the outcome of the request, the Finesse Library may need to do further processing. As I mentioned elsewhere (§3), the Finesse Library could have one of three outcomes:

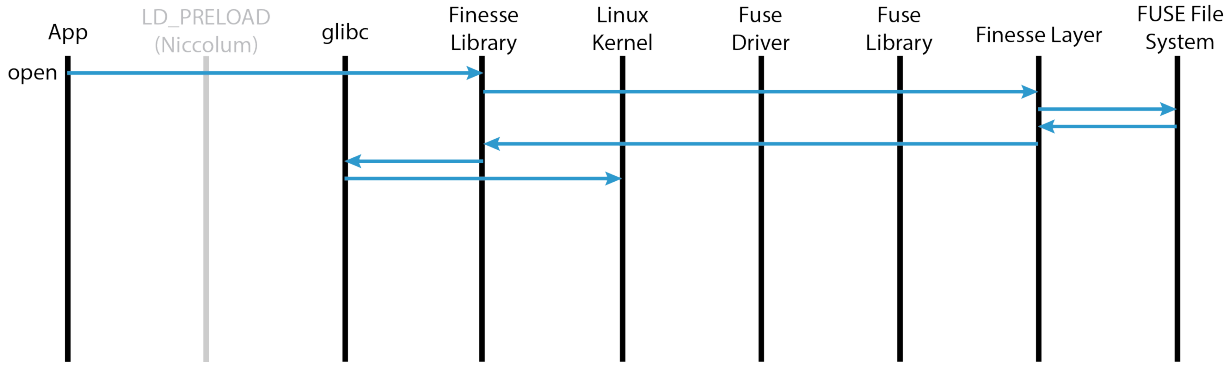
1. The request could be fatal. In that case an error is returned back to the application. This would happen if, for example, the file being opened does not exist.
2. The request could be a transient failure. For example, if the Finesse library is unable to message the FUSE/Finesse layer. In that case, the request should be passed to *glibc* for default processing.
3. The request could succeed.

The following diagram shows the success path. In this case, we still need to have a file handle, which is created by the kernel. Thus, we forward the request to *glibc* for processing.



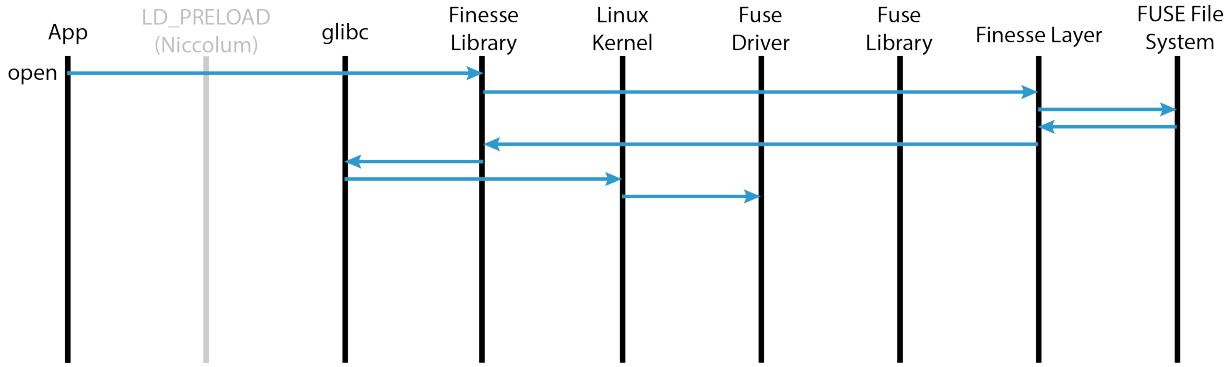
Finesse Library invokes *glibc* to obtain a handle (file descriptor) for the file.

glibc implements the *open* system call by invoking the Linux kernel.



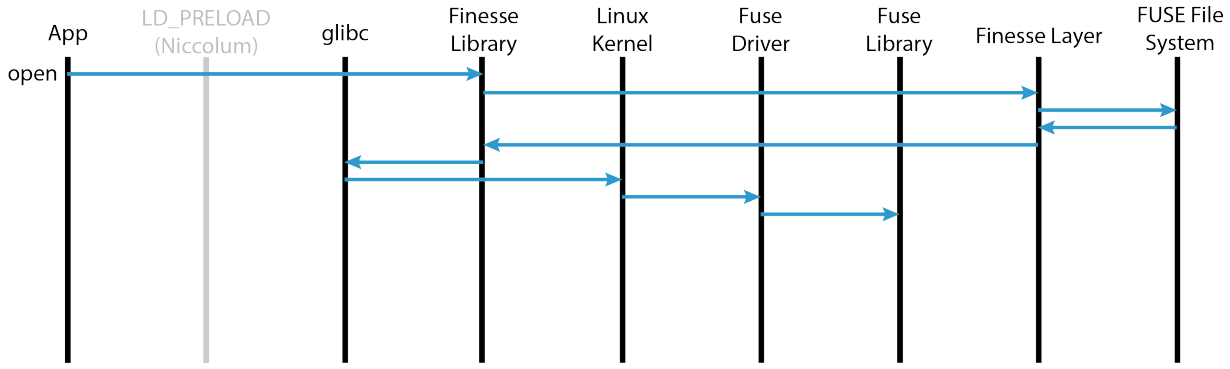
glibc invokes the *open* system call, which is implemented by the Linux kernel.

The Linux kernel parses the name and finds the mounted FUSE file system. The request is passed to the FUSE kernel driver, via the VFS interface.



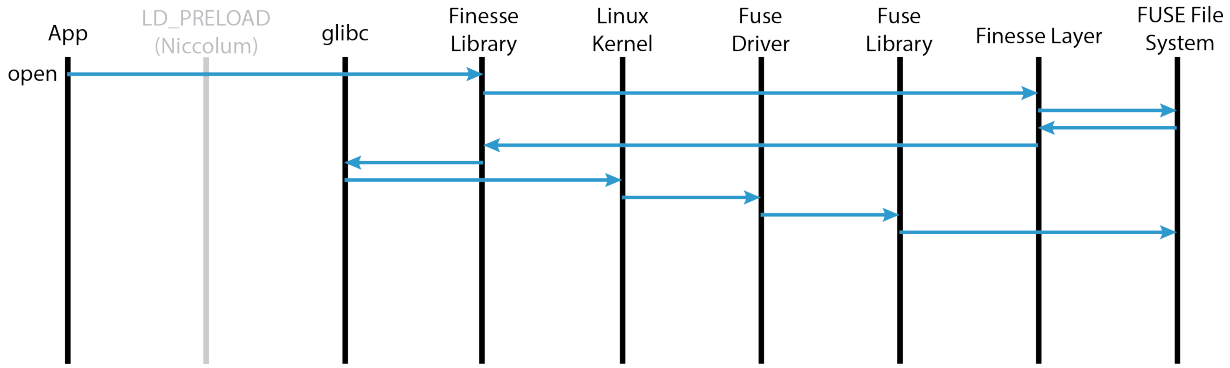
Linux kernel invokes FUSE kernel driver (via VFS)

The FUSE kernel driver formats a FUSE message (request) and enqueues it for the FUSE user mode library; the FUSE user mode library retrieves it via an *ioctl* call.



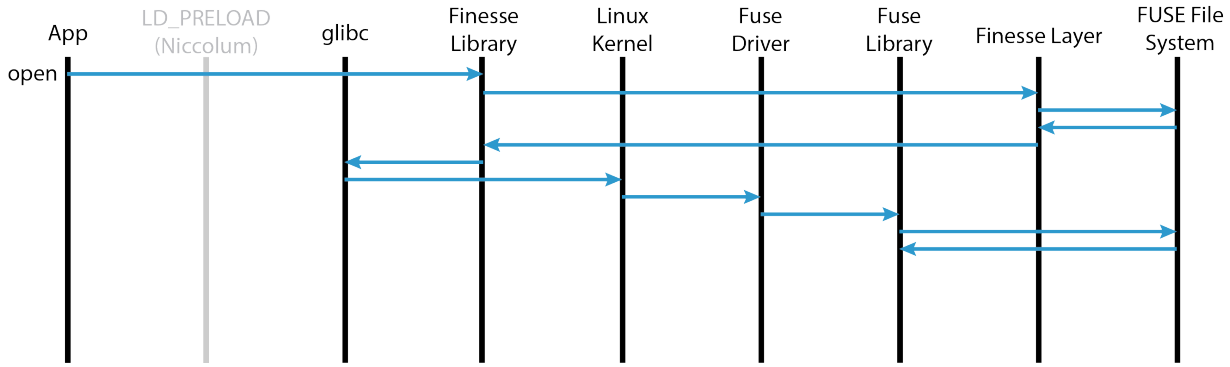
The FUSE kernel driver sends a request to the relevant FUSE library for handling.

Once the FUSE library has the request, it processes it. In this case it would make a *lookup* call to convert name into an inode number. Note: this is an iterative interface, so the kernel may invoke the FUSE file system multiple times as it processes individual component names within the path; I omit these multiple calls.



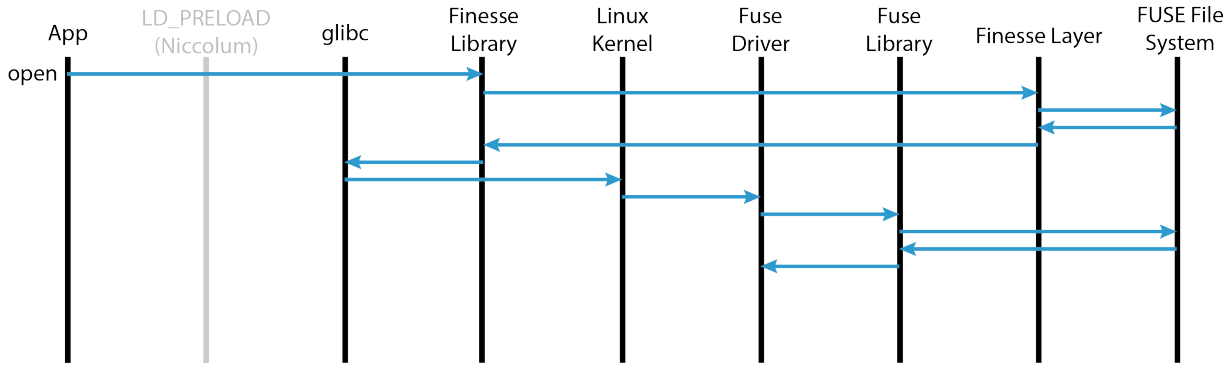
The FUSE library performs a *lookup* call into the FUSE file system.

The FUSE file system performs whatever processing it needs to in order to convert the name into an inode number. If the name is not found or there is some other error, an inode number is not returned.



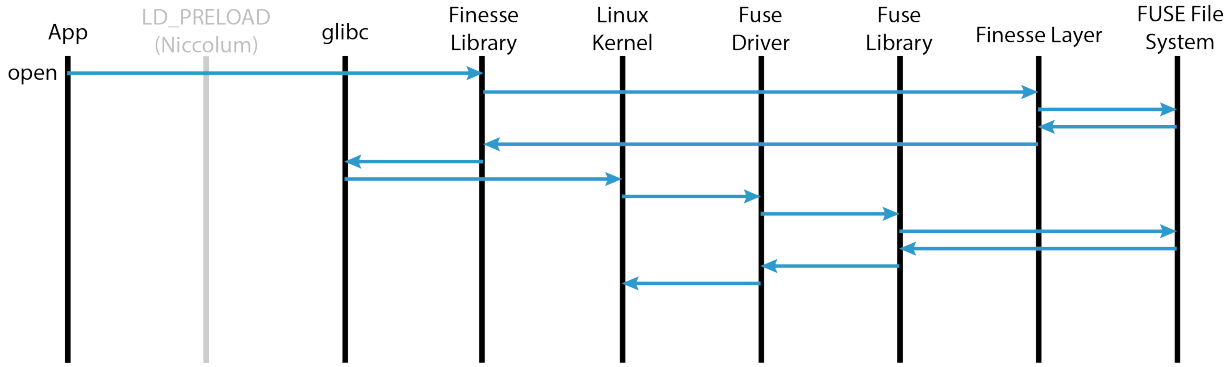
The FUSE file system returns the results of the *lookup* call back to the FUSE library.

Once the FUSE library receives the response to the *lookup* it formats a response message and sends it to the FUSE kernel driver.



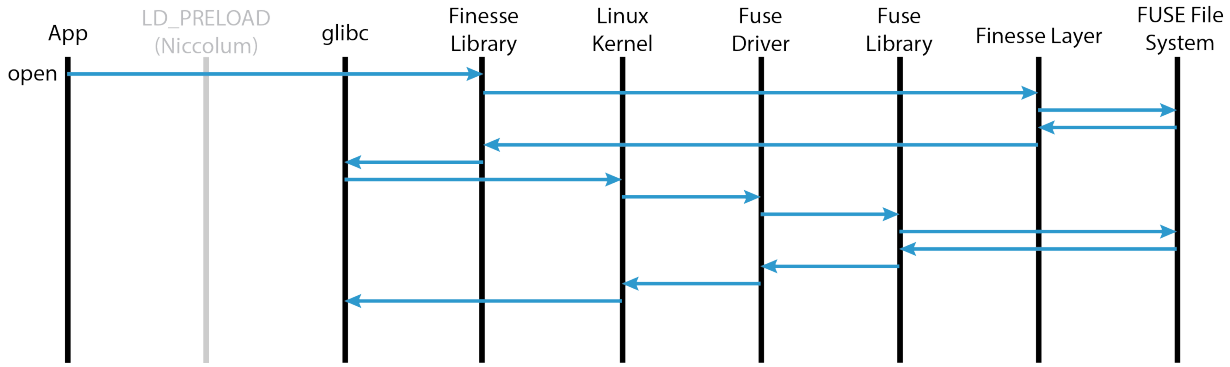
The FUSE library sends a response message to the FUSE kernel driver.

The FUSE kernel driver picks up the response, matches it to the request, and then, in turn, completes the original request from the Linux kernel.



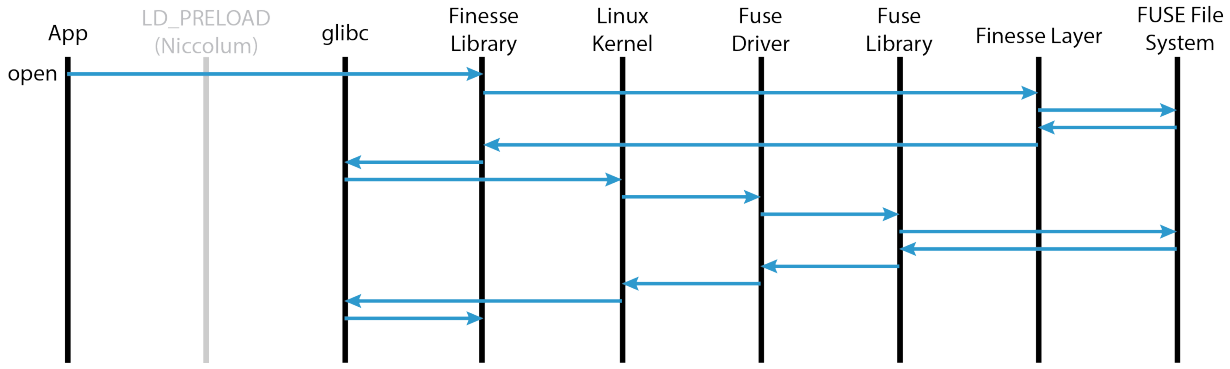
FUSE kernel driver sends the result to the Linux kernel.

The Linux kernel processes the response. This may involve creating a new file handle (the success path) or it may cause invocation of the VFS interface again (e.g., symbolic links or mount points). I'm not sure if you can mount on a user mode file system (it seems like a bad idea) but that might interfere in interesting ways with Finesse.



Linux kernel returns the results to the *glibc* library call.

glibc receives a response from the system call and returns the results back to the Finesse library.

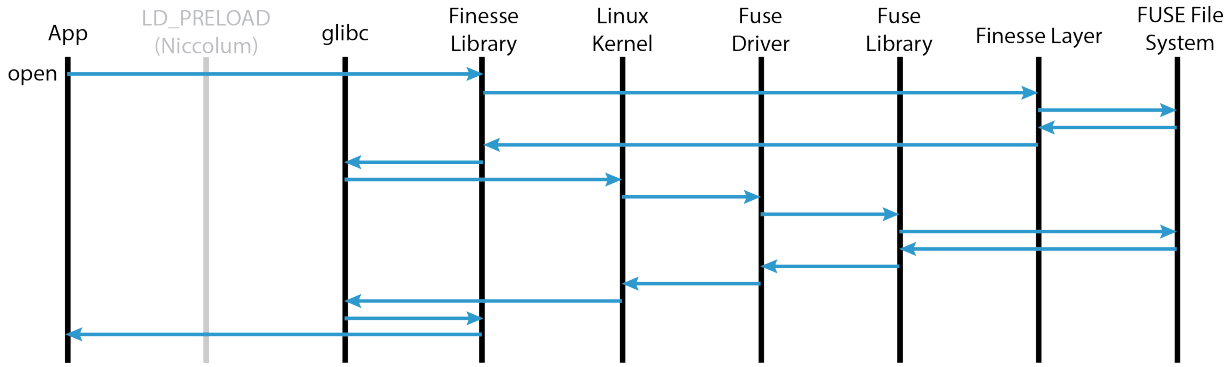


glibc returns results to Finesse Library.

Once the handle is returned from *glibc*, the Finesse Library does its bookkeeping to associate the given handle (file descriptor) with the UUID. This permits subsequent requests to send messages to the FUSE/Finesse layer specific to this particular file.

Note that there are potentially unexpected failure conditions here, such as the open call failing. In that case the Finesse Library needs to unwind any partially constructed state.

Finally, the Finesse Library returns the results to the application, along with the expected handle (or error status in case of failure).



Finesse Library returns results to application

The flow of other commands would be similar; note that there is no guarantee the calls map one-to-one. Finesse may not invoke *glibc* at all, if it is not necessary. Similarly, it may invoke different system calls in *glibc*, such as `lseek` to set the offset pointer, which is maintained in the kernel and associated with the underlying file handle. Note that multiple file *descriptors* may map to the same underlying kernel structure (e.g., if a handle is duplicated or passed between processes using UNIX domain sockets).

3 Preload

The function of LD_PRELOAD libraries are to intercept calls intended for an existing shared library and provide an alternative implementation of those calls. This functionality is commonly used across operating systems; LD_PRELOAD is the mechanism used on Linux to implement this functionality.

The model for this is quite simple: an existing call invokes our alternative implementation. For example, the `open` call is passed to our preload library. That implementation then determines if the call is directed to a Finesse-aware FUSE file system. If it is **not** then the call is passed to the next

registered handler for the `open` call, e.g., the implementation in *glibc*. The application is unaware of this logic.

Note that if the call is not passed to Finesse, no state is stored. Any subsequent handle-based operation will bypass Finesse because we will not be able to lookup any Finesse-relevant state for that file.

If the open request is passed to a Finesse-aware FUSE file system, this routine then calls the Finesse library handler for open. Note that there are three possible returns in this situation:

1. The open is successful and a file handle is returned to the caller; that file handle can subsequently be used to lookup Finesse state.
2. The open is *unsuccessful* and this is a permanent failure. In that case, no file handle is returned, *errno* is set appropriately, and the LD_PRELOAD handler will return a failure back to the application.
3. The open is *unsuccessful* but this is not a permanent failure (e.g., it is a failure within the Finesse handling itself). In that case, the call is passed to the next registered handler, just as if this request were not directed to a Finesse-aware FUSE file system. No state is stored.

In general, this path is expected to be fairly "light-weight". It is always supposed to be transparent to the application program.

Most (if not all) of the logic necessary for Finesse handling is encapsulated within the Finesse Library, not the LD_PRELOAD library.