

FUSE File System in User Space

<https://www.kernel.org/doc/Documentation/filesystems/fuse.txt>

CS333

Williams College

Announcements

- CS Movie night (the Matrix)
 - October 1: 6:30pm in Wege (with Pizza)
- WSO Engineering hours: Saturday 4-5pm CS common room (sometimes with Pizza)

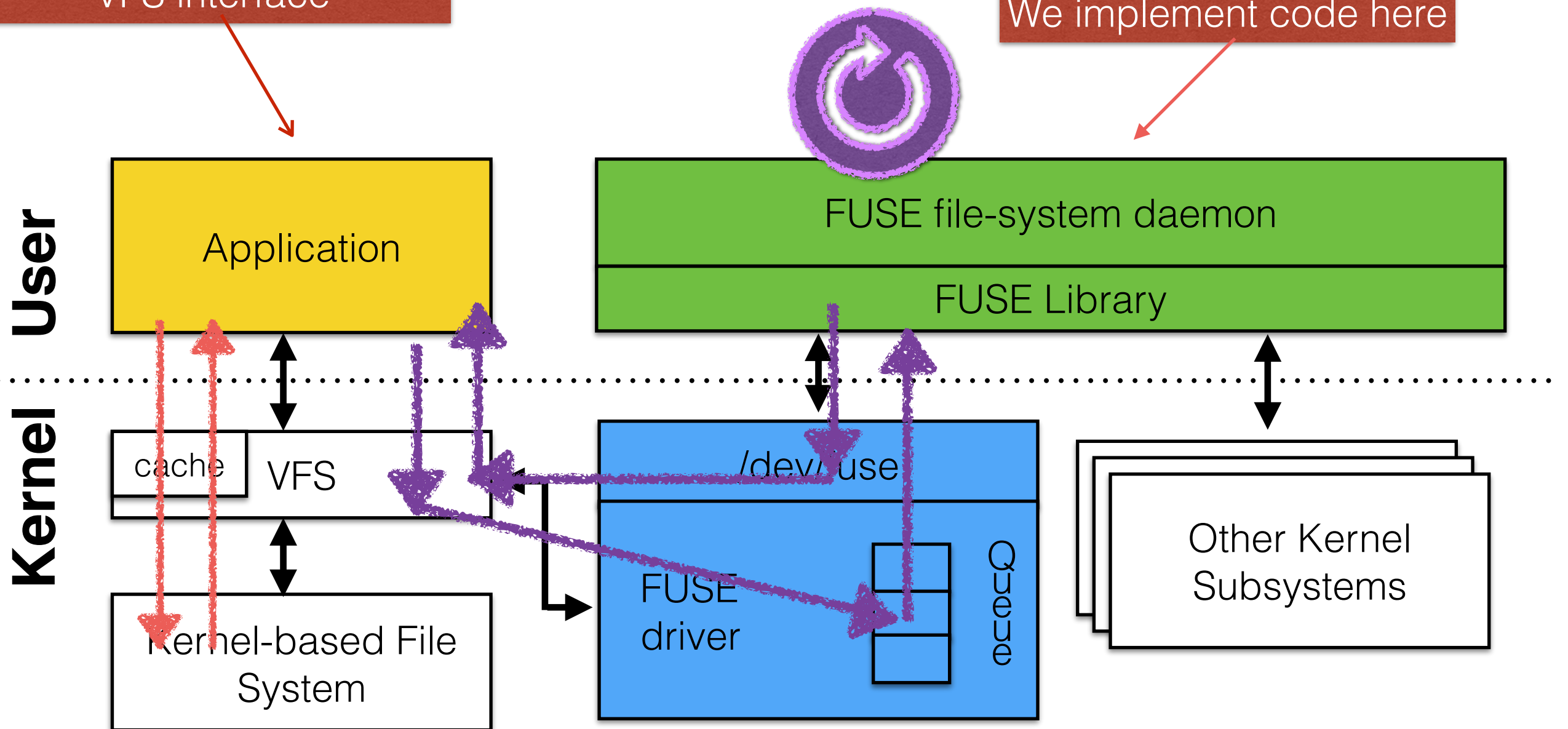
FUSE Overview

- The FUSE framework lets application writers create `userspace` file systems
 - + Faster and more convenient development
 - + Write and debug code using familiar tools
 - - Performance is often slower
- FUSE Consists of three parts:
 - FUSE kernel module
 - **libfuse**
 - **fusermount**

Fuse Kernel Module

Applications use standard VFS interface

We implement code here



“Userspace side” of FS Implementation

- `libfuse` provides an implementation of the FUSE interface for communicating with the FUSE kernel module.
 - `libfuse` v3 broke backwards compatibility; we'll be using v2.9.9 (the default version on Ubuntu 24.04)
 - Look at appropriate “git tag” to see correct source code
- By following templates in the `examples/` directory, we can see what infrastructure “simple” file systems require
- ***General idea***: similar to implementing a “real” file system
 - implement well-defined functions
 - register a C struct that holds our function pointers so our code is called by FUSE daemon (hooks)

Fusermount

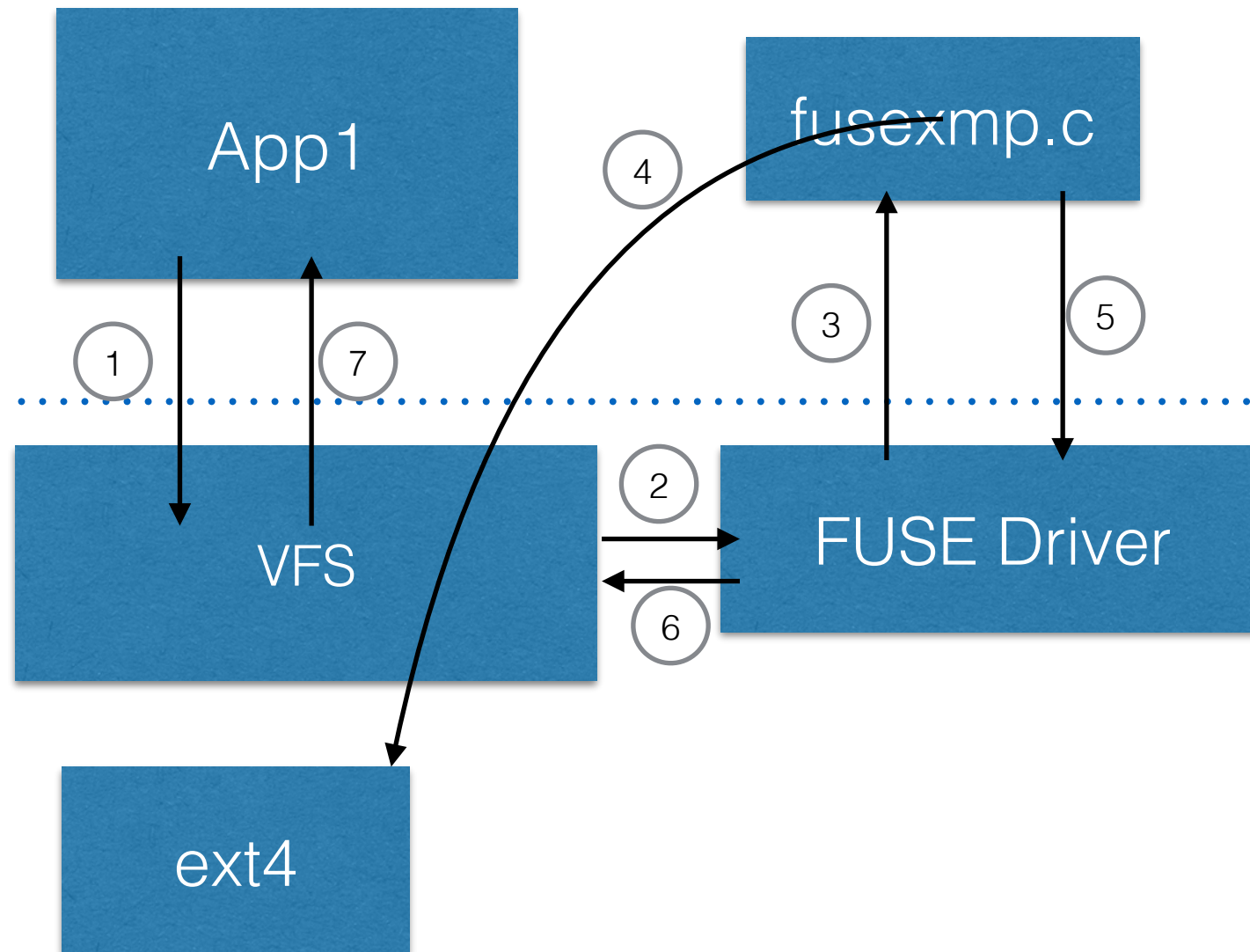
- Mounting a file system typically requires root privileges
- Users can mount/unmount their own FUSE filesystems.
 - The **fusermount** program is installed *setuid root*.
- However, there are restrictions:
 - You can only mount a FUSE FS on a mountpoint where your user has write permission
- Notes on general workflow:
 - Running your compiled FUSE FS program will mount it
 - Using the standard **mount** utility lists all mounted file systems, including FUSE file systems
 - **fusermount -u mntpnt** can unmount your FUSE FS

Common Uses for FUSE FSes

- “Pass-through” file system
 - fusexmp.c
- “Pseudo” file system
 - hello.c
- Prototypes/proof-of-concept designs
 - TokuFS
- Adding functionality that would be hard to provide inside the kernel
 - E.g., FS that relies on user-space libraries or APIs

Example: “fusexmp.c” Pass-through FS

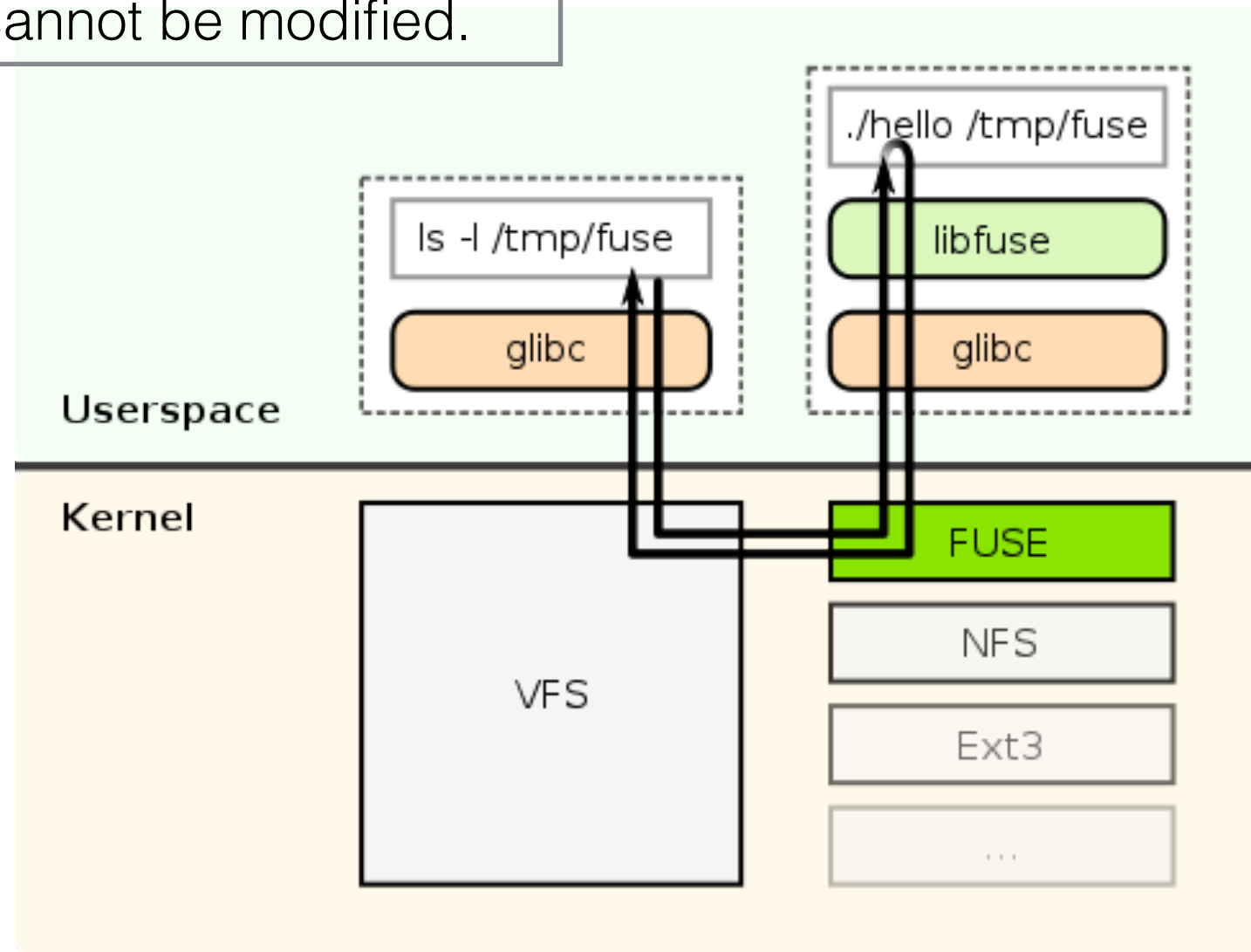
Goal: pass operations through to lower FS, (ext4) as if the FUSE FS was not there



- 1 App1 performs an operation on a file
- 2 The request is sent to the FUSE driver
- 3 The FUSE driver calls the corresponding function in fusexmp.c
- 4 fusexmp performs the requested operation on the underlying file system (here, ext4).
- 5 When fusexmp is done, it returns from the FUSE function to the FUSE Driver
- 6 The FUSE driver returns back to the VFS
- 7 The VFS returns back to the application

Example: “hello.c” In-memory FS

Goal: simulate a single read-only file.
The file is called “hello”; it does not exist on disk and it cannot be modified.

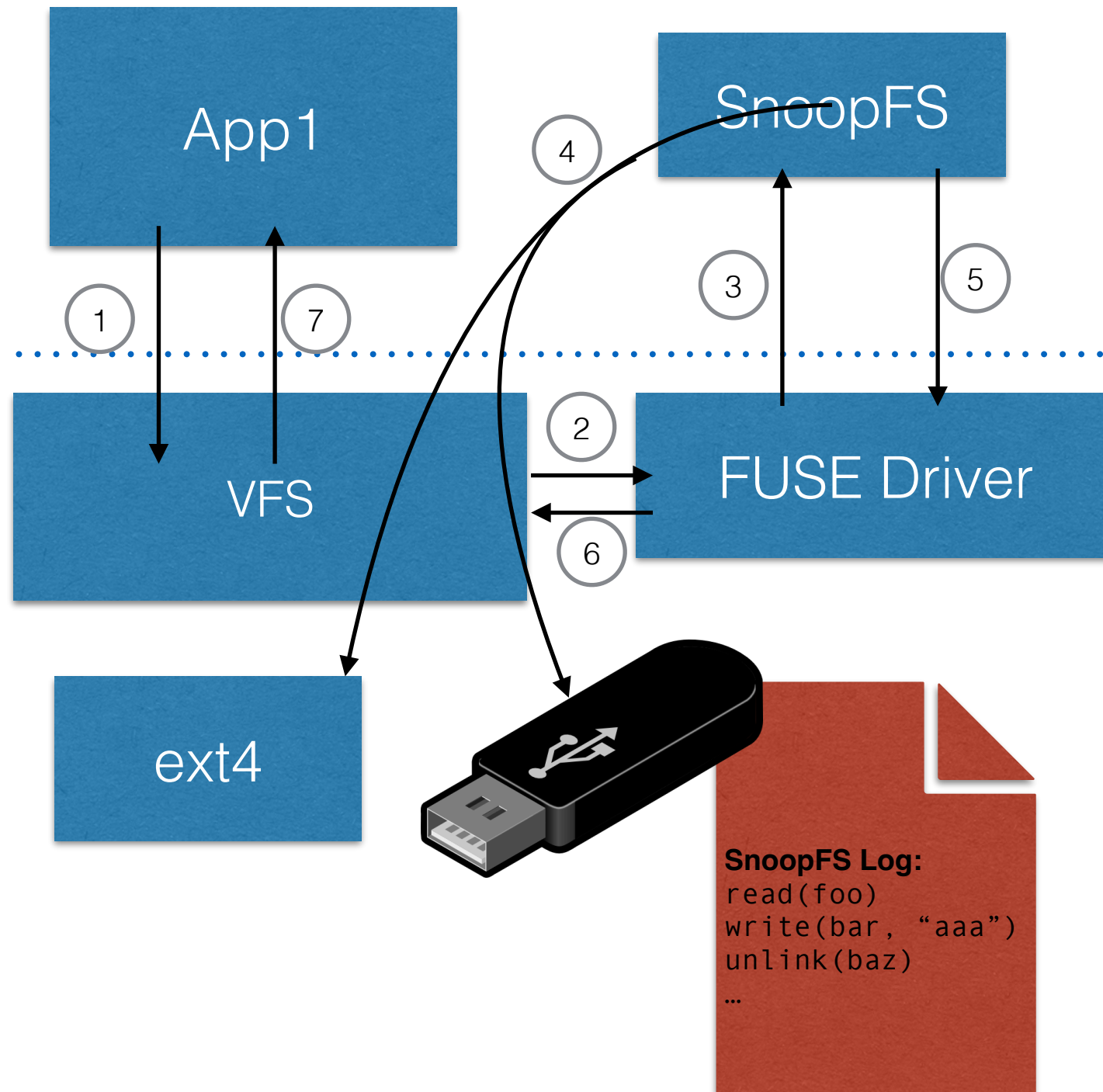


FUSE Demo

- Compile a FUSE FS
 - Remember to use appropriate gcc flags
- Run the FS to mount it
 - In example, the argument is the mount point
- Example usage
 - Operations on your file system are translated into calls to your FUSE functions
- Example Debugging with gdb
 - (in gdb) run with command ``run -s -d <mntpnt>``
- Unmount using **fusermount**
 - Use the `-u` flag

Example: "SnoopFS"

Goal: pass operations through to lower FS, (ext4) but track all operations on thumb drive



- 1 App1 performs an operation on a file
- 2 The request is sent to the FUSE driver
- 3 The FUSE driver calls the corresponding function in our SnoopFS
- 4 SnoopFS does whatever it wants. Here:
 - perform the actual operation (to avoid detection)
 - secretly log the operation on our hidden thumb drive
- 5 When SnoopFS finishes, it returns from the FUSE function
- 6 The FUSE driver returns back to the VFS
- 7 The VFS returns back to the application