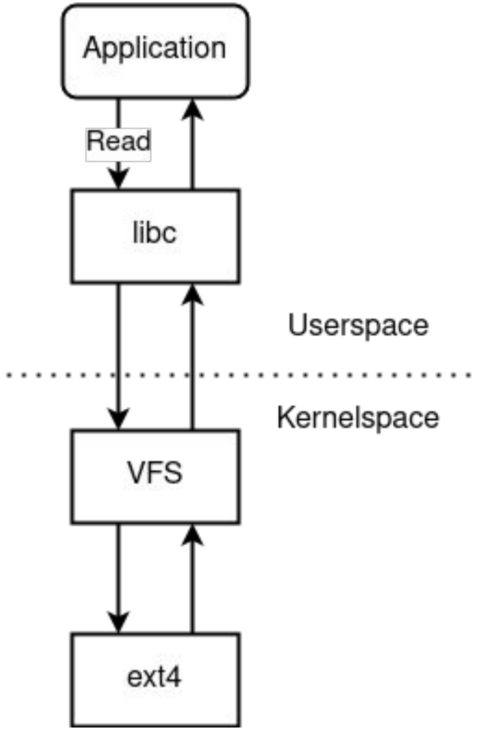
Introduction in FUSE filesystems

Charalampos Mainas

What is FUSE

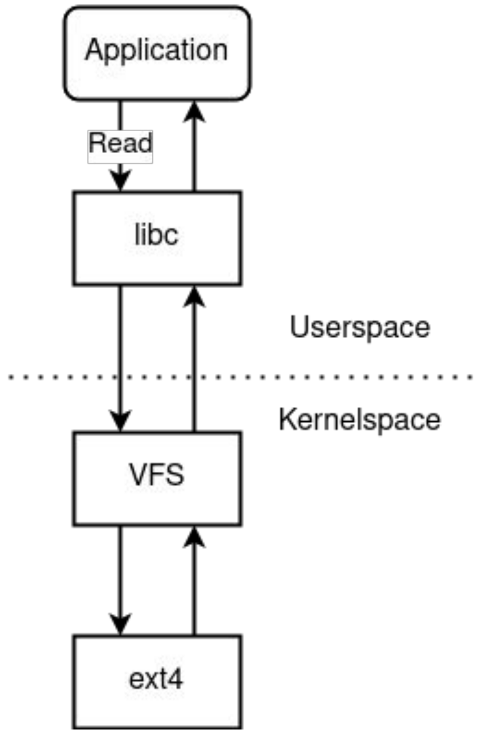
- A recap of filesystems
  - Part of the OS
  - An abstraction layer
  - Decides how data are stored and retrieved on disks
- FUSE: Filesystem in Userspace
  - Software layer in Userspace
  - Allows the creation of custom filesystems
  - Kernel code stays untouched

How does FUSE work

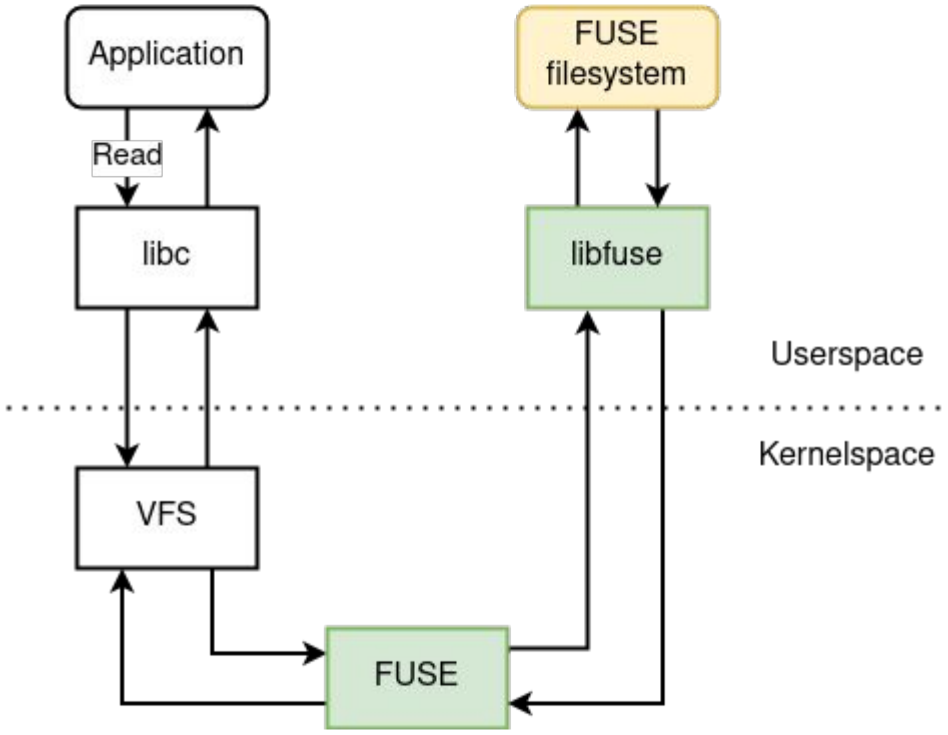


Normal filesystem

How does FUSE work



Normal filesystem



FUSE filesystem

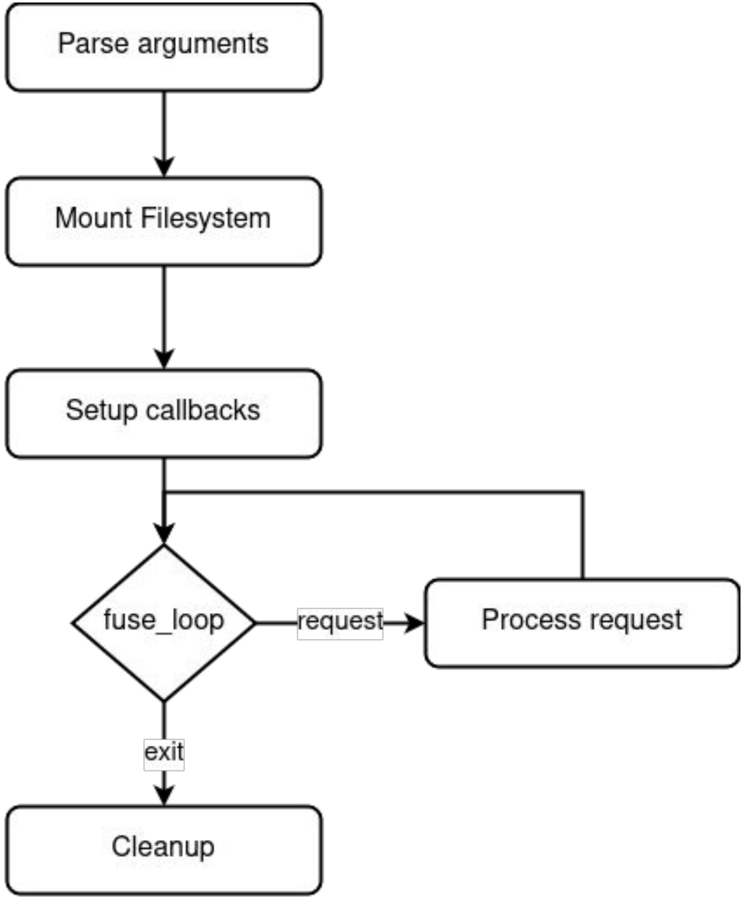
FUSE use cases

- Why a filesystem in userspace?
  - Shorter development cycle
  - Easier development, OS-agnostic
  - Does not affect other parts/services
  - Safer usage of untrusted filesystems
- Real world use cases:
  - On-disk filesystems: NTFS, retro-fuse etc.
  - Network-based filesystems: MinFS, SSHFS, etc.
  - Layering filesystems: EncFS, FuseCompress etc.
  - Archive, backup filesystems: Atlas, Borg etc.

How to use FUSE

- A FUSE application is a typical user-space program
  - Applications define how to handle filesystem operations
  - Interaction with libfuse to register the operations
  - Libfuse invokes the application defined operation upon a request
- API from libfuse
  - Callback mechanism for binding user-defined functions with operations
  - High-level API -- path level, synchronous
  - Low-level API – inode level, asynchronous

FUSE application workflow



The FUSE API

- FUSE allow the user to specify how a file operation will get handled
  - User implements a set of functions to handle a file operation
  - User fills the *struct fuse_operations* or *struct fuse_lowlevel_ops* with the respective function implementations
  - User passes the struct to libfuse and upon a request libfuse calls the respective function

```
static struct fuse_operations my_fuse_ops = {
        .getattr    = my_fuse_getattr,
        .read       = my_fuse_read,
        .write      = my_fuse_write,
        .readdir    = my_fuse_readdir,
        .mkdir      = my_fuse_mkdir,
        .mknod      = my_fuse_mknod,
        .open       = my_fuse_open,
        .create     = my_fuse_create,
        .readlink   = my_fuse_readlink,
        .symlink    = my_fuse_symlink,
};
```

FUSE operations

- lookup (only in the low-level API):
    - Searches the directory entry specified by last parameter and returns its attributes
- getattr:
    - Get the attributes of a file
- read:
    - Reads data from an open file
- write:
    - Writes data to an open file
- readdir:
    - Read a directory
- mkdir:
    - Create a directory

FUSE operations

- open:
  - Opens a file
- mknod:
  - Create a file node
- create:
  - Create and open a file
- symlink:
  - Create a symbolic link
- readlink:
  - Read the target of a symbolic link

Error messages

- In case a request fails an appropriate error should get returned
- Useful error codes:
  - ENOSYS: Function not implemented
  - EPERM: Operation not permitted
  - EACCESS: Permission denied
  - ENOENT: No such file or directory
  - EIO: I/O error
  - EEXIST: File exists
  - ENOTDIR: Not a directory
  - ENOTEMPTY: Directory not empty

Task assignment

- Implement a in-memory filesystem using FUSE
  - Everything is stored in the process's memory
  - Choose the high or low level API of FUSE
  - Implement at least the previously mentioned operations
  - Maximum file name length of 255 ascii characters.
  - Maximum file size of 512 bytes.

Useful tips

- Documentation for low-level API: https://libfuse.github.io/doxygen/fuse__lowlevel_8h.html
- Documentation for high level API: https://libfuse.github.io/doxygen/fuse_8h.html
- FUSE can parse command line arguments
  - Check fuse_parse_cmdline (low level API) and fuse_main (High level API)
- Do not get scared with mounting the filesystem
  - libfuse will mount it for you
    - fuse_mount for low level API
    - fuse_main for high level API
- Use -f option to execute your application in foreground
  - Useful when debugging, especially with printfs
- Unmount a mounted FUSE filesystem with
  - fusermount -u <mountpoint>

Thank you for listening!

see you in the Q&A session