CS-492 Course Project: File System

Due date May, 18

# Project Description

In this project, you will implement a disk-based file system called FSX492, which is a simple derivation of the Unix Fast File System (<https://pages.cs.wisc.edu/~remzi/OSTEP/file-ffs.pdf>). You will use the FUSE toolkit to implement the file system as a user-space process. Instead of a physical disk, you will be provided a file-based virtual disk, which can be accessed through a customized block device interface.

# Technical Background

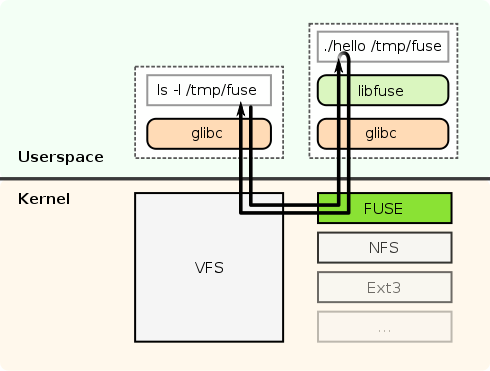


Figure 1: Architecture of FUSE

FUSE (File System in User Space) is a kernel module and library which allow you to implement POSIX file systems within a user-space process. Figure 1 shows the architecture of FUSE. All operations to the FUSE file system can be intercepted by callbacks placed by users through the FUSE interfaces. For instance, you can mount a “disk” to “/whatever/path/you/like”, in the format of a FUSE file system. Any operations you issued to this file system (e.g., open/read/write/close to directories or files under “/whatever/path/you/like”) will be redirected to FUSE (“libfuse” in Figure 1), where you can intercept the operations by registering callbacks to FUSE.

# Specifics of FSX492

We have prepared the specifics of FSX492, to facilitate your implementation.

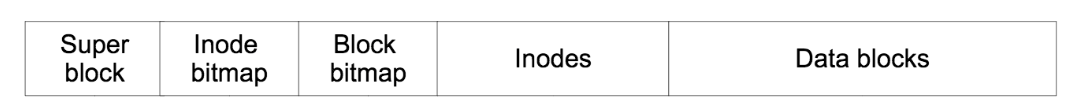


Figure 2: Format of the file-based virtual disk

* **File-based virtual block storage device.**

We have prepared a virtual block storage device within a single file on the host file system. FSX492 will run on this virtual block storage device. This device is divided into blocks of **1024** bytes, and into five regions (see Figure 2): the superblock, bitmaps for allocating inodes and data blocks, the inode table, and the rest of the blocks, which are available for storage for files and directories.

* **Superblock**

The superblock is the first block in the file system, and contains the information needed to find the rest of the file system structures. The following C constants and structure can be used to implement the superblock:

---------------------------------------------------

enum {

FS\_BLOCK\_SIZE = 1024, /\* block size in bytes \*/

FS\_MAGIC = 0x37363030 /\* magic number for superblock \*/

};

struct fs\_super {

uint32\_t magic; /\* magic number \*/

uint32\_t inode\_map\_sz; /\* inode map size in blocks \*/

uint32\_t inode\_region\_sz; /\* inode region size in blocks \*/

uint32\_t block\_map\_sz; /\* block map size in blocks \*/

uint32\_t num\_blocks; /\* total blocks, including SB, bitmaps, inodes \*/

uint32\_t root\_inode; /\* always inode 1 \*/

char pad[FS\_BLOCK\_SIZE - 6 \* sizeof(uint32\_t)]; /\* pad out to an entire block \*/

}; /\* total FS\_BLOCK\_SIZE bytes \*/

---------------------------------------------------

Note that uint16\_t and uint32\_t are standard C types found in the <stdint.h> header file, and refer to unsigned 16 and 32-bit integers. (similarly, int16\_t and int32\_t are signed 16 and 32-bit integers.

* **Inodes**

These are standard Unix-style inodes. Each inode corresponds to a file or directory; in a sense the inode is that file or directory, which can be uniquely identified by its inode number. The root directory is always found in inode 1; inode 0 is reserved (this allows ‘0’ to be used as a null value). The following C structure can be used to implement an inode:

---------------------------------------------------

enum {N\_DIRECT = 6 }; /\* number direct entries \*/

struct fs\_inode {

uint16\_t uid; /\* user ID of file owner \*/

uint16\_t gid; /\* group ID of file owner \*/

uint32\_t mode; /\* permissions | type: file, directory, ... \*/

uint32\_t ctime; /\* creation time \*/

uint32\_t mtime; /\* last modification time \*/

int32\_t size; /\* size in bytes \*/

uint32\_t direct[N\_DIRECT]; /\* direct block pointers \*/

uint32\_t indir\_1; /\* single indirect block pointer \*/

uint32\_t indir\_2; /\* double indirect block pointer \*/

uint32\_t pad[3]; /\*padding to make 64 bytes per inode \*/

}; /\* total 64 bytes \*/

---------------------------------------------------

Up to INODES\_PER\_BLK inodes can be stored in a block (of size FS\_BLOCK\_SIZE):

---------------------------------------------------

enum {INODES\_PER\_BLK = FS\_BLOCK\_SIZE/sizeof(struct fs\_inode) };

---------------------------------------------------

The direct array and the indir\_1 and indir\_2 fields store block numbers that “point” to blocks of storage. The blocks pointed to by direct array elements are blocks in the file. The blocks pointed to by the indir\_1 and indir\_2 fields are blocks that contain arrays of block numbers. For indir\_1, those block numbers point to file blocks. For indir\_2, those block numbers point to other blocks that contain arrays of block numbers that point to file blocks (we do not require more than 2-layers of indirections). There are PTRS\_PER\_BLK 32-bit block numbers per block, giving a maximum file size of about 64MB.

enum {PTRS\_PER\_BLK = FS\_BLOCK\_SIZE / sizeof(uint32\_t) };

* **Directories**

Directories are one block in length, which limits the size of a directory to **32 entries**. This will simplify your implementation quite a bit, as you can allocate that block in ‘mkdir’ and not have to worry about extending it when you are adding a new entry. Directory entries are quite simple, with two flags indicating whether an entry is valid or not and whether it is a directory, an inode number, and a name. Note that the maximum name length is 27 bytes, allowing entries to always have terminating 0 bytes.

The following C constant and structure can be used to implement a directory entry:

---------------------------------------------------

enum {FS\_FILENAME\_SIZE = 28 }; /\* max file name length \*/

struct fs\_dirent {

uint32\_t valid : 1; /\* entry valid flag; “1” means 1 bit \*/

uint32\_t isDir : 1; /\* entry is directory flag; “1” means 1 bit\*/

uint32\_t inode : 30; /\* entry inode; “30” means 30 bit \*/

char name[FS\_FILENAME\_SIZE]; /\* with trailing NUL \*/

}; /\* total 32 bytes \*/

---------------------------------------------------

* **Storage allocation**

Inodes and blocks are allocated by searching the respective bitmaps for entries which are cleared. Note that in the file-based device we prepared, the blocks used for the superblock, bitmaps, and inodes are marked as in-use, so you don’t have to worry about avoiding them during allocation. Inodes 0 and 1 are marked, as well. Bitmaps are stored in as many blocks as required for the number of inodes and blocks (which can be learnt from the superblock). There are BITS\_PER\_BLK bits per block.

enum {BITS\_PER\_BLK = FS\_BLOCK\_SIZE \* 8 };

The number of bitmap blocks allocated for inodes is calculated as the *number of inode blocks* times *INODES\_PER\_BLK*, divided by *BITS\_PER\_BLK*, rounded up to the nearest block. This is stored in the superblock inode\_map\_sz field. A set of macros provided with the C select() function can be used to access individual bits in the inode and block bitmaps that are read into memory when the disk is mounted. The macros are in <select.h> or <sys/select.h> . Here is an example of how to use them to set, query, and clear the bit for an inode in the inode block map

---------------------------------------------------

fd\_set \*inode\_map = malloc(sb.inode\_map\_sz \* FS\_BLOCK\_SIZE);

/\*read inode bitmap from disk image into ‘inode\_map’ … code skipped\*/

uint32\_t inum = 100; /\* number for inode 100 \*/

FD\_SET(inum, inode\_map); /\* mark inode as in use \*/

if (FD\_ISSET(inum, inode\_map)) { … } /\* is inode in use? \*/

FD\_CLR(inum, inode\_map); /\* mark inode as free \*/

---------------------------------------------------

# Tasks

Your main tasks are to develop code to support block-level I/O operations to the file-based virtual block storage device and develop code to support common operations to the files and directories in the file system.

To ease your implementations, we have prepared a skeleton of FSX492 at <https://classroom.github.com/a/SVAgFRNU>. “Task 3: Testing” gives instructions for building and running the code skeleton (before you implement the actual code, the skeleton can do nothing).

## Task 1: Block-level I/O Interfaces **(100pts)**

You will need to implement the POSIX block device I/O interfaces that will be used by the FSX492 file system:

---------------------------------------------------

static int num\_blocks(struct blkdev \*dev)

/\*

count the number of blocks on the device

@param dev: the block device

@return: the number of blocks in the block device

\*/

static int image\_read(struct blkdev dev, int first\_blk, int nblks, void\* buf)

/\*

read blocks from block device starting at give block index

@param dev: the block device

@param first\_blk: index of the block to start reading from

@param nblks: number of blocks to read from the device

@param buf: buffer to store the data

@return: return status of the operation (success or error reason)

\*/

static int image\_write(struct blkdev dev, int first\_blk, int nblks, void \*buf)

/\*

write bytes to block device starting at give block index

@param dev: the block device

@param first\_blk: index of the block to start writing to

@param nblks: number of blocks to write to the device

@param buf: buffer where data comes from

@return: return status of the operation (success or error reason)

\*/

static int image\_flush(struct blkdev \*dev, int first\_blk, int nblks)

/\*

Flush the block device.

@param dev: the block device

@aparam first\_blk: index of the block to start flushing

@param nblks: number of blocks to flush

@return: return status of the operation (success or error reason)

\*/

static void close(struct blkdev \*dev)

/\*

close the device. After this, access to device will return errors

@param dev: the block device

\*/

---------------------------------------------------

To help your implementation and reduce the difficulty to get started, we have prepared a source code file “image.c” in the above mentioned code skeleton.

In “image.c”, we have also implemented the following function:

---------------------------------------------------

struct blkdev \*image\_create(char \*path)

---------------------------------------------------

This function will create and return a block device, which provides an abstraction of the file-based disk (at the location pointed to by path). All the above to-be-implemented functions are attached as interfaces of the created device, which you can use to operate on the device. You should read through the code of image\_create function to get a better sense.

In “image.c”, we have also created placeholders for the above function. What you have to do is to implement the placeholders following the above requirements.

## Task 2: File/Directory Management and Operations **(100pts)**

You will need to implement the following functions to support operations to the FSX492 file system:

---------------------------------------------------

void\* fs\_init(struct fuse\_conn\_info \*conn)

/\*

init - this is called once by the FUSE framework at startup.

This is a good place to read in the super-block and set up any

global variables you need. You don't need to worry about the

argument or the return value.

@param conn: fuse connection information - unused

@return: unused - returns NULL

\*/

static int fs\_getattr(const char \*path, struct stat \*sb)

/\*

getattr - get file or directory attributes. For a description of

the fields in 'struct stat', see 'man lstat'.

Note - you can handle some fields as follows:

* st\_nlink - always set to 1
* st\_atime, st\_ctime - set to same value as st\_mtime

@param path: the file path

@param sb: pointer to stat struct

@return: 0 if successful, or -error number

* -ENOENT - a component of the path is not present
* -ENOTDIR - an intermediate component of path not a directory

\*/

static int fs\_opendir(const char \*path, struct fuse\_file\_info \*fi)

/\*

open - open file directory

You can save information about the open directory in fi->fh. If you allocate memory, free it in fs\_releasedir.

@param path: the file path

@param fi: fuse file system information

@return: 0 if successful, or -error number

* -ENOENT - a component of the path is not present
* -ENOTDIR - an intermediate component of path not a directory

\*/

static int fs\_readdir(const char \*path, void \*ptr, fuse\_fill\_dir\_t filler, off\_t offset, struct fuse\_file\_info \*fi)

/\*

readdir - get directory contents

For each entry in the directory, invoke the 'filler' function, which is passed as a function pointer, as follows:

filler(buf, <name>, <statbuf>, 0) where <statbuf> is a struct stat, just like in getattr.

@param path: the directory path

@param ptr: filler buf pointer

@param offset: the file offset -- unused

@param fi: the fuse file information -- you do not have to use it

@return: 0 if successful, or -error number

* -ENOENT - a component of the path is not present
* -ENOTDIR - an intermediate component of path not a directory

\*/

static int fs\_releasedir(const char \*path, struct fuse\_file\_info \*fi)

/\*

Release resources when directory is closed.

If you allocate memory in fs\_opendir, free it here.

@param path: the directory path

@param fi: fuse file system information -- you do not have to use it

@return: 0 if successful, or -error number

* -ENOENT - a component of the path is not present
* -ENOTDIR - an intermediate component of path not a directory

\*/

static int fs\_mknod(const char \*path, mode\_t mode, dev\_t dev)

/\*

mknod - create a new file with permissions (mode & 01777). Behavior undefined when mode bits other than the low 9 bits are used.

@param path: the file path

@param mode: indicating block or character-special file

@param dev: the character or block I/O device specification - you do not have to use it

@return: 0 if successful, or -error number

* -ENOTDIR - component of path not a directory
* -EEXIST - file already exists
* -ENOSPC - free inode not available
* -ENOSPC - results in >32 entries in directory

\*/

static int fs\_mkdir(const char \*path, mode\_t mode)

/\*

mkdir - create a directory with the given mode. Behavior undefined when mode bits other than the low 9 bits are used.

@param path: path to directory

@param mode: the mode for the new directory

@return: 0 if successful, or -error number

* -ENOTDIR - component of path not a directory
* -EEXIST - file already exists
* -ENOSPC - free inode not available
* -ENOSPC - results in >32 entries in directory

\*/

static int fs\_unlink(const char \*path)

/\*

unlink - delete a file

@param path: path to file

@return 0 if successful, or error value

* -ENOENT - file does not exist
* -ENOTDIR - component of path not a directory
* -EISDIR - cannot unlink a directory

\*/

static int fs\_rmdir(const char \*path)

/\*

rmdir - remove a directory

@param path: the path of the directory

@return: 0 if successful, or -error number

* -ENOENT - file does not exist
* -ENOTDIR - component of path not a directory
* -ENOTDIR - path not a directory
* -ENOEMPTY - directory not empty

\*/

static int fs\_rename(const char \*src\_path, const char \*dst\_path)

/\*

rename - rename a file or directory. You can assume the destination and the source share the same path-prefix

Note that this is a simplified version of the UNIX rename

functionality - see 'man 2 rename' for full semantics. In

particular, the full version can move across directories, replace a destination file, and replace an empty directory with a full one.

@param src\_path: the source path

@param dst\_path: the destination path

@return: 0 if successful, or -error number

* -ENOENT - source file or directory does not exist
* -ENOTDIR - component of source or target path not a directory
* -EEXIST - destination already exists
* -EINVAL - source and destination not in the same directory

\*/

static int fs\_chmod(const char \*path, mode\_t mode)

/\*

chmod - change file permissions

@param path: the file or directory path

@param mode: the mode\_t mode value -- see man 'chmod'

for description

@return: 0 if successful, or -error number

* -ENOENT - file does not exist
* -ENOTDIR - component of path not a directory

\*/

static int fs\_open(const char \*path, struct fuse\_file\_info \*fi)

/\*

Open a filesystem file or directory path.

@param path: the path

@param fuse: file info data

@return: 0 if successful, or -error number

* -ENOENT - file does not exist
* -ENOTDIR - component of path not a directory

\*/

static int fs\_read(const char \*path, char \*buf, size\_t len, off\_t offset, struct fuse\_file\_info \*fi)

/\*

read - read data from an open file.

@param path: the path to the file

@param buf: the buffer to keep the data

@param len: the number of bytes to read

@param offset: the location to start reading at

@param fi: fuse file info

@return: return exactly the number of bytes requested, except:

- if offset >= file len, return 0

- if offset+len > file len, return bytes from offset to EOF

- on error, return <0

* -ENOENT - file does not exist
* -ENOTDIR - component of path not a directory
* -EIO - error reading block

\*/

static int fs\_write(const char \*path, const char \*buf, size\_t len, off\_t offset, struct fuse\_file\_info \*fi)

/\*

write - write data to a file

@param path: the file path

@param buf: the buffer to write

@param len: the number of bytes to write

@param offset: the offset to starting writing at

@param fi: the Fuse file info for writing

@return: It should return exactly the number of bytes requested, except on error:

* -ENOENT - file does not exist
* -ENOTDIR - component of path not a directory
* -EINVAL - if 'offset' is greater than current file length. (POSIX semantics support the creation of files with "holes" in them, but we don't)

\*/

static int fs\_release(const char \*path, struct fuse\_file\_info \*fi)

/\*

Release resources created by pending open call.

@param path: path to the file

@param fi: the fuse file info

@return: 0 if successful, or -error number

* -ENOENT - file does not exist
* -ENOTDIR - component of path not a directory

\*/

static int fs\_statfs(const char \*path, struct statvfs \*st)

/\*

statfs - get file system statistics. See 'man 2 statfs' for description of 'struct statvfs'.

@param path: the path to the file

@param st: pointer to the destination statvfs struct

@return: 0 if successful, or -error number

* -ENOENT - a component of the path is not present
* -ENOTDIR - an intermediate component of path not a directory

\*/

---------------------------------------------------

To help your implementation and reduce the difficulty to get started, we have prepared a source code file “fs.c” in the above mentioned code skeleton.

In “fs.c”, we have prepared a global data structure fs\_ops, which has already included the above functions as hooks. This data structure is used in “main.c” (prepared by us) to register those hooks. That said, the following operations to the FSX492 file system will be redirected to the above functions you implemented, in the following way:

---------------------------------------------------

init --> fs\_init,

getattr --> fs\_getattr

opendir --> fs\_opendir

readdir --> fs\_readdir

releasedir --> fs\_releasedir

mknod --> fs\_mknod

mkdir --> fs\_mkdir

unlink --> fs\_unlink

rmdir --> fs\_rmdir

rename --> fs\_rename

chmod --> fs\_chmod

open --> fs\_open

read --> fs\_read

write --> fs\_write

release --> fs\_release

statfs --> fs\_statfs

---------------------------------------------------

In “fs.c”, we have also created placeholders for the above function. What you have to do is to implement the placeholders following the above requirements.

## Task 3: Testing (Everything Needs to be Done in the VM) **(60pts)**

We have prepared a virtual block device at “test/fsx492.img” (in the repo we prepared). You can follow the instructions below to test your implementation:

### Step 1: install FUSE

* $: sudo apt-get update
* $: sudo apt-get install fuse libfuse-dev

### Step 2: Build the FSX492

* Fork the code skeleton from the github classroom and enter the code repo
* $: make (this will produce a binary called fsx492)

### Step 3: Test FSX492

There are two modes to test FSX492: **command line mode** and **FUSE mode.**

In command line mode you are provided with an FTP-like interface which allows you to run the interfaces you developed and modify the block storage device; *this mode may be easily run under a debugger.*

The FUSE mode, in turn, allows the block storage device to be mounted as a standard Linux file system, on which you can use shell commands to operate.

Note: backup the virtual block device before running any tests

**Command line mode:**

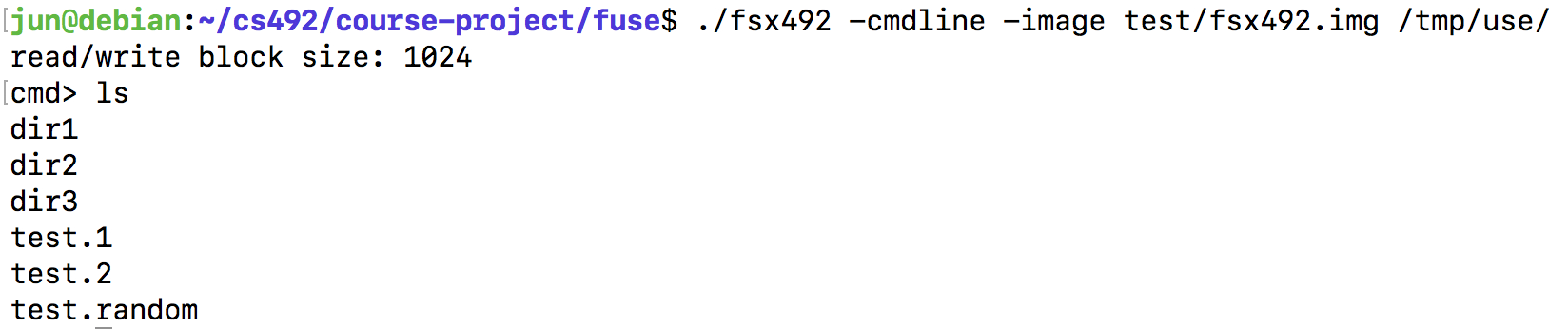
1. Create a new, empty folder, e,.g., /tmp/use/
2. Start FSX492 under command line mode: $ ./fsx492 -cmdline -image test/fsx492.img /tmp/use/
   1. If everything goes well, you will be able to enter a console showing “cmd>”, where you can go to step 3.
3. You can run the following commands in the console created above.

---------------------------------------------------

* "cd” - change to root directory
* "cd <dir>” - change to directory at <dir>
* "pwd" - display current directory
* "ls" - list files in current directory
* "ls <dir>” - list specified directory
* "ls-l” - display detailed file listing
* "ls-l <file>” - display detailed file info of <file>
* "chmod <mode> <file>” - change permissions of <file> to <mode>
* "rename <oldname> <newname>” - rename file from oldname to newname
* "mkdir <dir>” - create directory <dir>
* "rmdir <dir>” - remove directory <dir>
* "rm <file>” - remove file <file>
* "put <outside> <inside>” - copy a file from host directory <outside> into FSX492 at <inside>
* "get <inside> <outside>” - retrieve a file from FSX492 at <inside> to host directory at <outside>
* "show <file>” - retrieve and print a file at <file>
* "statfs” - print file system info
* "touch <file>” - create file or set modified time to current time
* "stat <file>” - print info of <file>

---------------------------------------------------

For instance, if you run “ls”, you suppose to see:

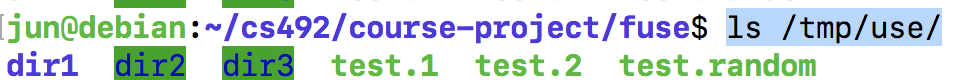


This command line mode is very convenient for debugging. If you look into the code, you will be able to see that the above commands are backed with functions you will implement. Also, you can start this mode inside GDB and trace the execution for actions like single-step debugging.

**The FUSE mode:**

1. Create a new, empty folder, e,.g., /tmp/use/
2. Start FSX492 under FUSE mode and mount the virtual block device to the above folder: $ ./fsx492 -image test/fsx492.img /tmp/use/
   1. If everything goes well, you will not see any errors and you can go to step 3.
3. Now you can run all kind of file/directory operations to /tmp/use/ using shell commands

For instance, if you run “ls /tmp/use/”, you suppose to see:



If you want to unmount the file system: you can run “fusermount -u /tmp/use”

## Task 4: Project Report **(40pts)**

You will need to write a report about the course project. The report will need to include the following sections:

* Design and implementation of the file system
* Design and implementation of testing
* Group information
  + Members
  + Contribution of each member

## Policy about Grouping

* You are allowed to group together for this course project
* You are allowed to have at most 4 students in your group
* You are allowed to team up with students from different sections