

For this homework we are going to implement the fsx600 file system, a simple derivative of the Unix FFS file system. We will use the FUSE toolkit in Linux to implement the file system as a user-space process; instead of a physical disk we will use a data file accessed through a block device interface specified at the end of this file. (the blkdev pointer can be found in the global variable 'disk') This document describes the file system and the FUSE toolkit; a separate PDF file is posted describing the details of the assignment itself.

fsx600 File System Format

The disk is divided into blocks of 1024 bytes, and into five regions: 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.

Super block	Inode bitmap	Block bitmap	Inodes	Data blocks
-------------	--------------	--------------	--------	-------------

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 structure can be used for the superblock:

```
struct fsx_superblock {
    uint32_t magic;
    uint32_t inode_map_size;    /* in 1024-byte blocks */
    uint32_t inode_region_sz;  /* in 1024-byte blocks */
    uint32_t block_map_sz;     /* in 1024-byte blocks */
    uint32_t num_blocks;        /* total disk size */
    uint32_t root_inode;
    padding[];                  /* to make size = 1024 */
};
```

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 ints)

Inodes

These are standard Unix-style inodes with single and double-indirect pointers, for a maximum file size of about 64MB. 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, e.g. in direct and indirect pointers)

uint16_t uid	File owner
uint16_t gid	Group
uint16_t mode	Permissions and type
uint32_t ctime uint32_t mtime	Timestamps
uint32_t size	Of file, in bytes
uint32_t direct[6]	First 6 blocks in file
uint32_t indir_1	Height-1 tree of pointers
uint32_t indir_2	Height-2 tree of pointers

Directories

Directories are a multiple of one block in length. You are free to limit the size of a directory to one block, which simplifies your implementation quite a bit. (you can allocate the block in 'mkdir' and not worry about it) This limits the number of entries in a directory to 32, and means that size=1024.

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.

valid : 1 bit
isDir : 1 bit
inode : 30 bits
name: 28 bytes

Storage allocation

Inodes and blocks are allocated by searching the respective bitmaps for entries which are cleared. Note that when the file system is first created (by mktest or the mkfs-hw3 program) the blocks used for the superblock, maps, 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.

The following functions are provided at the top of homework.c to access these bitmaps:

```
bit_set(map, i);  
bit_clear(map, i);  
bit_test(map, i);
```

where 'map' is a pointer to the memory containing the bitmap and 'i' is the index to set, clear, or check.

FUSE API

FUSE (File system in USEr space) is a kernel module and library which allow you to implement Linux file systems within a user-space process. For Homework 3 we will use the C interface to the FUSE toolkit to create a program which can read, write, and mount fsx600 file systems. When you run your working program, it should mount its file system on a normal Linux directory, allowing you to 'cd' into the directory, edit files in it, and otherwise use it as any other file system.

To write a FUSE file system you need to:

1. define file methods – `mknod`, `mknod`, `delete`, `read`, `write`, `getdir`, ...
2. register those methods with the FUSE library
3. call the FUSE event loop

FUSE Data structures

The following data structures are used in the interfaces to the FUSE methods:

path – this is the name of the file or directory a method is being applied to, relative to the mount point. Thus if I mount a FUSE file system at `"/home/pjd/my-fuseFS"`, then an operation on the file `"/home/pjd/my-fuseFS/subdir/filename.txt"` will pass `"/subdir/filename.txt"` to any FUSE methods invoked.

Note that the 'path' variable is read-only, and you have to copy it before using any of the standard C functions for splitting strings. (`strtok` or `strsep`)

mode – when file permissions need to be specified, they will be passed as a *mode_t* variable: owner, group, and world read/write/execute permissions encoded numerically as described in 'man 2 chmod'.¹

device – several methods have a 'dev_t' argument; this can be ignored.

struct stat – described in 'man 2 lstat', this is used to pass information about file attributes (size, owner, modification time, etc.) to and from FUSE methods.

struct fuse_file_info – this gets passed to most of the FUSE methods, but we don't use it.

Error Codes

FUSE methods return error codes standard UNIX kernel fashion – positive and zero return values indicate success, while a negative value indicates an error, with the particular negative value used indicating the error type. The error codes you will need to use are:

EEXIST – a file or directory of that name already exists

ENOENT – no such file or directory

EISDIR, ENOTDIR – the operation is invalid because the target is (or is not) a directory

ENOTEMPTY – directory is not empty (returned by `rmdir`)

EOPNOTSUPP – operation (i.e. method) not supported. When you're done you won't use this..

ENOMEM, ENOSPC – operation failed due to lack of memory or disk space

¹ Special files (e.g. `/dev` files) are also indicated by additional bits in a mode specifier, but we don't implement them in fsx600.

In each case you will return the negative of the value; e.g.:

```
return -ENOENT; /* file not found */
```

A note on permissions

For simplicity we are not checking permissions on files, as it is rather difficult to test easily. (in particular, FUSE runs as your user ID, so creating files with other user IDs and testing access is a bit difficult.) You are responsible for storing ‘mode’ (the file permission field) correctly and returning it via the appropriate interfaces (getattr, readdir) but your code will never return EACCESS.

FUSE Methods

The methods that you will have to implement are listed below. Note that some of the possible error codes are due to path translation – e.g. if file.txt exists, then 'mkdir /file.txt/dirname' will fail with ENOTDIR. See comments in homework.c for more details.

- mkdir(path, mode) – create a directory with the specified mode. Returns success (0), EEXIST, ENOENT or ENOTDIR if the containing directory can't be found or is a file. Remember to set the creation time for the new directory. You'll need to OR 'mode' with S_IFDIR
- rmdir(path) - remove a directory. Returns success, ENOENT, ENOTEMPTY, ENOTDIR.
- create(path, mode, info) – create a file with the given mode. Ignore the 'info' argument. Return values are success, EEXIST, ENOTDIR, or ENOENT. Remember to set the creation time for the new file. (you **don't** need to OR 'mode' with S_IFREG, although it won't hurt)
- unlink(path) - remove a file. Returns success, ENOENT, or EISDIR.
- readdir - read a directory, using a rather complicated interface including a callback function. See the sample code for more details. Returns success, ENOENT, ENOTDIR.
- getattr(path, attrs) – returns file attributes. (see 'man lstat' for more details of the format used)
- read(path, buf, len, offset)– read 'len' bytes starting at offset 'offset' into the buffer pointed to by 'buf'. **Returns the number of bytes read on success** - this should always be the same as the number requested unless you hit the end of the file. If 'offset' is beyond the end of the file, return 0 – this is how UNIX file systems indicate end-of-file. Errors – ENOENT or EISDIR if the file cannot be found or is a directory.
- write(path, buf, len, offset)– write 'len' bytes starting at offset 'offset' from the buffer pointed to by 'buf'. **Returns the number of bytes written on success** - this should always be the same as the number requested. If 'offset' is greater than the current length of the file, return EINVAL.² Errors: ENOENT or EISDIR. Remember to set the modification time.

² UNIX file systems support “holes”, where you can write to a location beyond the end of the file and the region in the middle is magically filled with zeros. We don't.

`truncate(path, offset)` – delete all bytes of a file after 'offset'. If 'offset' is greater than zero, return `EINVAL`³; otherwise delete all data so the file becomes zero-length. Remember to set the modification time.

`rename(path1, path2)` – rename a file or directory. If 'path2' exists, returns `EEXISTS`. If the two paths are in different directories, return `EINVAL`. (these are both “cheating”, as proper Unix file systems are supposed to handle these cases)

`chmod(path, mode)` – change file permissions.

`utime(path, timebuf)` – change file modification time. (the `timebuf` structure also contains an access time field, but we don't use it)

`statfs(path, statvfs)` – returns statistics on a particular file system instance – fill in the total number of blocks and you should be OK.

Note that in addition to any error codes indicted above in the method descriptions, the 'write', 'mkdir', and 'create' methods can also return `ENOSPC`, if they are unable to allocate either a file system block or a directory entry.

For full Linux compatibility you should update the modified timestamp on the containing directory when performing create, unlink, rename, mkdir, and rmdir, but we won't be testing that.

Path translation

You should use a helper function to do path translation. Note that this function must be able to return multiple error values – consider the following paths in a standard Unix file system:

`/usr/bin/cat/file.txt`
`/usr/bin/bad-file-name`

In the first case you would need to return `-ENOTDIR`, as `/usr/bin/cat` is a regular file and so directory traversal cannot proceed. In the second case you would return `-ENOENT`, as there is no entry for 'bad-file-name' in the `/usr/bin` directory. **NOTE** – this means that any method except `statfs` can return `ENOENT` or `ENOTDIR` due to a failure in path translation.

Hints, additional information, and shortcuts

timestamps – file system timestamps are in units of seconds since sometime in 1970. To get the current time, use `'time(NULL)'`.

directory entries and **directory blocks** – when you are factoring your code, remember that bitmaps, inodes, and directories have multiple entries in a single disk block, so you have to make sure you don't accidentally erase the ones you are not changing.

Attribute translation

When implementing `getattr` and `readdir` you will need to translate the information available in your file system to a `'struct stat'` passed by the FUSE framework. The `uid`, `gid`, `mode`, `ctime`, `mtime`, and `size` fields translate directly. The other entries in the `stat` buffer can be set to zero. (hint – `'memset(sb, 0,`

³ UNIX allows truncating a file to a non-zero length, but this is rarely used so we skip it.

sizeof(sb))' before setting those values and you'll be OK.)

The 'mode' field

This field holds additional attributes besides the permissions. The primary one we have to worry about is the flag indicating the object type – i.e. regular file or directory. (there are other types for device files, symbolic links, etc. but we don't implement them.)

S_IFDIR (octal 0040000) indicates a directory. It can be tested for with the macro 'S_ISDIR(mode)'

S_IFREG (octal 0100000) indicates a regular file, and can be tested for with 'S_ISREG(mode)'

Debugging

Error – “Transport endpoint is not connected” - this happens when your FUSE process crashes. You'll still need to unmount the directory that it was mounted on:

```
fusermount -u directory/
```

Factoring

You will probably want to factor out **at least** the following functionality:

1. path translation. There are two cases:
 - “/a/b/c” → inode number for ‘c’ (for most operations)
 - “/a/b/c” → inode number for ‘b’ + leaf name “c” (for mknod, mkdir, etc.)
2. translate block offset in file to block number on disk. (for read)
3. allocate block at offset in file (for write)

Note that even though #2 and #3 are specific to particular functions, if you factor them out the higher-level logic will be much clearer and easier to debug (e.g. read and mkdir)

Unsigned integers

Don't use them. There is going to be a lot of logic using negative values to indicate errors, and if you put those into an unsigned variable they'll become positive.

Disk images

Rather than a real device, we'll be using disk image files, and writing data blocks to the corresponding locations in that file. If you're brave, you can actually use mkfs-hw3 to format a flash drive and run your file system against the device file /dev/sdX. (if you're trying this, you'll know how to figure out which device file it is; you'll also have to disable the '.img' check in misc.c)