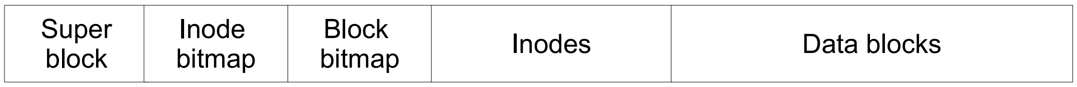
For this homework we are going to implement the fsx600 file system, a simple derivative of the Unix FFS file system. We will use the FUSE toolkit to implement the file system as a user-space process. Instead of a physical disk we will use a data file accessed through a block device interface specified 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 the assignment page on the course website describes 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.



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 ints.

Inodes

These are standard Unix-style inodes. Each 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)

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];                /* 64 bytes per inode */
};                                  /* total 64 bytes */
```

Up to INODES_PER_BLK inodes can be stored in a FS_BLOCK_SIZE block.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. 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. [larger directories are an exercise for the reader]

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 */
    uint32_t isDir : 1;                 /* entry is directory flag */
    uint32_t inode : 30;                /* entry inode */
    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 when the file system is first created (by mktest or the mkfs-x6 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. Bitmaps are stored in as many blocks as required for the number of inodes and blocks. 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 inode blocks time INODES_PER_BLK, divided by BITS_PER_BLK, rounded up to the 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' …
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 */
```

Block Storage Device

Our file system makes use of a virtual block storage device that provides storage within a single file on on the host file system. This enables controlling the creation and use of the storage, instead of using files and directories on the host file system. Our virtual file system runs on this virtual block storage device.

The block storage device provides block-level I/O operations that are used by our virtual file system. These operations correspond closely to POSIX block device I/O operations:

operation	description
num_blocks(struct blkdev *dev)	number of blocks in the block device
read(struct blkdev dev, int first_blk, int nblks, void* buf)	read bytes from block device starting at give offset
write(struct blkdev dev, int first_blk, int nblks, void *buf)	write bytes to block device starting at give offset
flush(struct blkdev *dev, int first_blk, int nblks)	flush block device
close(struct blkdev *dev)	close block device. After this, access to device will return E_UNAVAIL

The block device defines a struct that stores pointers to functions that implement these operations for a given implementation.

```
struct blkdev_ops {
   int  (*num_blocks)(struct blkdev *dev);
   int  (*read)(struct blkdev *dev, int first_blk, int num_blks, void *buf);
   int  (*write)(struct blkdev *dev, int first_blk, int num_blks, void *buf);
   int  (*flush)(struct blkdev *dev, int first_blk, int num_blks);
   void (*close)(struct blkdev *dev);
};
```

The block device is represented by a struct that provides access to the device operations and also carries private data used by the block device. The following C structure implements a block device:

```
struct blkdev {
   struct blkdev_ops *ops;              /* operations on block device */
   void *private;                       /* block device private state */
};
```

Our implementation of the block device implements these functions, and provides a function, image_create(char *path) that opens the image file, and returns a blkdev pointer for this device.

FUSE API

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. For assignment 4 we will use the C interface to the FUSE toolkit to create a program that can read, write, and mount fsx600 file systems. When you run your working program, it should mount this virtual file system on a normal 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, mkdir, delete, read, write, getdir, ...
2. register those methods with the FUSE library
3. call the FUSE event loop

Installing Fuse

On MacOS X, we recommend that you use the Brew package manager to install Fuse. This will install the files in */usr/local/{bin,lib,include}*.

On Linux, we recommend that you use the appropriate package manager (e.g. apt) for your flavour of Linux to install Fuse.

On Windows 10 under CygWin, we recommend installing the WinFsp package, which includes a Fuse compatible interface. Download the software at http://www.secfs.net/winfsp/download/ and review the installation instructions and release notes.

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. If a FUSE file system is mounted 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 not supported. You'll use this one a lot.
ENOMEM, ENOSPC – operation failed due to lack of memory or disk space

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

Fuse provides a *struct fuse_operations* with pointers to operations that must be implemented:

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.
rmdir(path)	remove a directory. Returns success, ENOENT, ENOTEMPTY, ENOTDIR.
create(path,mode,finfo)	create a file with the given mode. Ignore the 'finfo' argument. Return values are success, EEXIST, ENOTDIR, or ENOENT. Remember to set the creation time for the new file.
unlink(path)	remove a file. Returns success, ENOENT, or EISDIR.
Readdir(path,fdirht, finfo)	read a directory, using a rather complicated interface including a callback

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.

	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.
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 POSIX 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

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.

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

traversal cannot procede. 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.

FUSE Debugging

The driver code provided can be operated in two modes – command line mode and FUSE mode. In command line mode you are provided with an FTP-like interface which allows you to explore and modify a disk image; this mode may be easily run under a debugger. (to run the FUSE version under gdb, run with the '-d' and '-s' flags)

```
pjd@bubbles$ ./homework --cmdline disk1.img
cmd> ls
dir1  dir2  dir3
cmd> cd dir1
cmd> ls
file1.txt x   y    z
cmd> show file1.txt
[...]
cmd> quit
pjd@bubbles$
```

The file system interface, in turn, allows a fsx600 image to be mounted as a standard Linux file system. The two interfaces are shown below:

```
pjd@bubbles$ ./homework -image disk1.img mydir
pjd@bubbles$ ls mydir/
dir1  dir2  dir3
pjd@bubbles$ ls mydir/dir1
file1.txt  x  y  z
pjd@bubbles$ fusermount -u mydir
pjd@bubbles$ ls mydir
pjd@bubbles$
```

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, 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.

For your convenience, several functions have been suggested in the "homework.c" file that perform some of these operations. You are welcome to use them or delete them and create your own.

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.