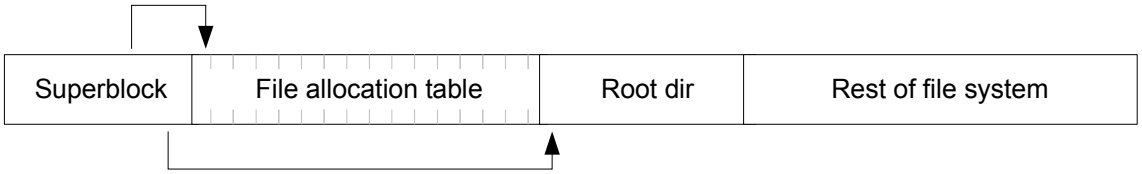
For this homework we are going to implement the cs5600fs file system, a simple derivative of the MSDOS FAT file system. We will use the FUSE toolkit in Linux to implement the file system as a user-space process; instead of a physical disk we will use a data file accessed through a block device interface 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.

cs5600fs File System Format

The disk is divided into blocks of 1024 bytes, and into three regions: the superblock, the file allocation table, and the actual storage for files and directories.



NOTE – you will be reading 512-byte disk sectors from the disk image, and will need to read 2 sectors for each file system block, and multiply all addresses by 2.

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. Note that information about the root directory is organized as a 'struct dirent', which you will find very convenient when you are writing your code.

Bytes 0-3	Magic Number (0xC0005600)
4-7	Block size (1024)
8-11	Total file system size (in blocks)
12-15	FAT length (in blocks)
16-80	Root directory dirent
	18-19: Uid (0)
	20-21: Gid (0)
	22-23: Mode (permissions)
	24-27: Modification time
	28-31: Start (block number)
30-1023	Unused

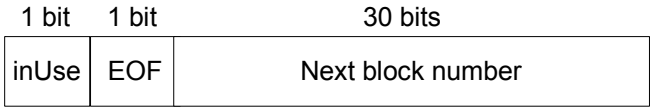
The following C structure can be used to implement the superblock:

```
struct cs5600fs_superblock {
        uint32_t magic;
        uint32_t block_size;   /* always 1024 */
        uint32_t filesys_size; /* in 1024-byte blocks */
        uint32_t fat_size;     /* in 1024-byte blocks */
        struct cs5600_dirent root_dirent;
};
```

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)

File Allocation Table:

This contains a 4-byte entry for every 1024-byte block in the file system; the entries look like this:



The FAT entries are used to chain blocks together in a directory or a file – i.e. if a file contains blocks 20, 21, and 30, then:

FAT[20] = {inUse=1, EOF=0, next=21}
FAT[21] = {inUse=1, EOF=0, next=30}
FAT[30] = {inUse=1, EOF=1, next=xxx}

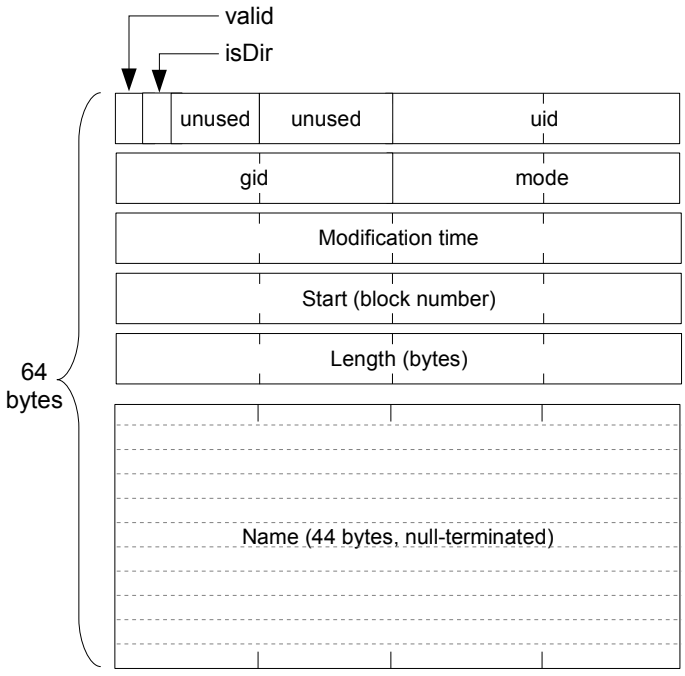
FAT entries for the superblock and the FAT itself are reserved by setting {inUse=1,eof=1}.

The structure for a FAT entry, using a C bitfield structure to access the in_use and EOF bits, is:

```
struct cs5600fs_fat {
        uint32_t inUse : 1;
        uint32_t eof : 1;
        uint32_t next : 30;
};
```

Directory:

Each directory entry is 64 bytes long, and contains the fields shown at right. The *uid*, *gid*, *mode*, and *modification time* fields are used by Linux, and we won't worry about them for now. The *valid, isDir, Start,* and *length* fields are important for your implementation.



The following C structure will implement a directory entry for you:

```
struct cs5600fs_dirent {
        uint16_t  valid : 1;
        uint16_t  isDir : 1;
        uint16_t  pad   : 6;
        uint16_t  uid;
        uint16_t  gid;
        uint16_t  mode;
        uint32_t  mtime;
        uint32_t  start;
        uint32_t  length;
```

```
        char name[44];
};
```

To simplify the assignment, directories are always 1 1024-byte block long, thus containing 16 directory entries each of 64 bytes. You can read a directory into an array of 16 directory entries as follows (note that a directory is 1 1024-byte file system block = 2 512-byte disk sectors):

```
struct cs5600fs_dirent directory[16];
disk->ops->read(disk, blkaddr*2, 2, (void*)directory);
```

Note that you cannot redefine the `cs5600fs_dirent` structure (e.g. to add extra fields to it) because it has to correspond to the layout of the data on the disk; you will be unable to read the disk images provided (and used for testing) if you modify this structure.

If you read a 1024-byte block into the buffer defined above, you can then iterate through it by linear search:

```
for (i=0; i<16; i++) {
      if (directory[i].valid) {
            do something...;
      }
}
```

For instance, to traverse a directory path 'do something' would compare directory[i].name to the current path component; for readdir it would translate the attributes into Linux format and call the filler function.

Storage allocation

Files consist of lists of blocks linked by their FAT pointers. It may be easiest to think of each block as a structure with 1024 bytes of data and a 30-bit *next* pointer – the next pointer corresponding to file system block *i* goes in entry *i* in the file allocation table.

To allocate a disk block, you search through the FAT entries until you find one with 'inUse==0'. To free a block, you set 'inUse=0' in the corresponding FAT entry.

Directories are fixed in size, and only span one block. To add an entry to a directory, you search through the directory until you find an entry with 'valid==0'; if not, then return the appropriate error. To remove an entry from a directory, just set 'valid=0' – there's no need to compact the directory or clear the old fields. When you're searching through a directory, remember that any entry can be valid – even if the 2nd through 15th entries are not valid, the 16th could have the entry you are searching for.

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 4 we will use the C interface to the FUSE toolkit to create a program which can read, write, and mount CS5600fs 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, mkdir, 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.

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 */
```

1 Special files (e.g. /dev files) are also indicated by additional bits in a mode specifier, but we don't implement them in cs5600fs.

A note on permissions

Due to the limited time available for this homework, no permission checks will be performed in this semester's version of cs5600fs; in particular, this means that your code will never return the EACCESS error code. Note, however, that you are responsible for storing 'mode' (the file permissions field) and returning it properly in the appropriate interfaces.

FUSE Methods

The methods that you will have to implement are:

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.

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.

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.

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.

rename(path1, path2) – rename a file or directory. If 'path2' exists, returns EEXISTS. If the two paths are in different directories, return EINVAL.

chmod(path, mode) – change file permissions.

utime(path, timebuf) – change file access & modification times.

statfs(path, statvfs) – returns statistics on a particular file system instance – block size, total/free/used block count, max name length. Always returns success.

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.

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.

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 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 of the preceding methods (except statfs) can return ENOENT or ENOTDIR due to a failure in path translation.

FUSE Debugging

The skeleton 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' flag) The file system interface, in turn, allows a cs5600fs image to be mounted as a standard Linux file system. The two interfaces are shown below:

```
pjd@bubbles$ ./on-disk 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$
```

File System Interface

```
pjd@bubbles$ ./on-disk --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$
```

Command line interface

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 for methods that modify the file system, you need to be able to read a directory block, find an entry in it, modify the entry, and then **write back the complete directory block** with the other entries unchanged.

order of implementation - The first operation I would suggest implementing is getattr; after that you should be able to run it in cmdline mode with disk1.img and do 'ls-l home':

```
pjd@bubbles:$ ./homework --cmdline disk1.img
cmd> ls-l home
/home drwxrwxr-x 0 0
cmd>
```

Then readdir, and you can list the root directory:

```
cmd> ls
home
work
dir_other
file.txt
another-file
```

Then add the read method, and you can print out a file from the image:

```
cmd> show file.txt
file.txt file.txt file.txt file.txt file.txt file.txt file.txt
file.txt
...
```

At this point you should be able to run it as a (read-only) FUSE file system. When you're debugging the FUSE version you should run it with the debug ('-d') option, as this prevents it from going into the background and makes it much easier to debug. (also you'll notice if it crashes)

Attribute translation

In cs5600fs, all file attributes are found in the directory entry:

- isDir – whether it's a file or a directory
- uid, gid – user and group ids. You get these from the OS (via FUSE) and give them back; you don't have to worry about their contents.
- mode – file permissions
- mtime – modification time
- length – file length in bytes (0 for a directory)

Note that 'valid' and 'start' are internal information, and don't translate into file or directory attributes.

You will need to translate between these values and Linux file attributes, which are found in 'struct stat'. (described in the manual page for the 'stat' system call: `man 2 stat`) The translation rules are

as follows:

- **st_dev, st_ino, st_rdev, st_blocks, st_blocksize, st_flags, st_gen**: ignore (linux → cs5600fs), set to 0 (cs5600fs → linux)
- **st_nlink**: ignore (linux → cs5600fs), set to 1 (cs5600fs → linux)
- **st_uid, st_gid, st_size** – copy directly
- **st_mtime** – copy directly
- **st_atime, st_ctime** – ignore (linux → cs5600fs), set to mtime (cs5600fs → linux)
- **mode** – this is the tricky one, as Linux uses it to indicate both permissions and whether an entry is a file or a directory. For linux → cs5600fs, just copy it. For the other direction, use the following code fragment:

```
mode = dirent->mode | (dirent->isDir ? S_IFDIR : S_IFREG)
```

Safeguards in misc.c

There are two checks in misc.c which are intended to prevent certain errors during debugging:

- Disk image files must end in '.img'. The sample files in your Git repositories end in '.img.orig', so you'll have to make a copy before you run your tests. That way you won't accidentally corrupt a test image and check it back in. The test scripts make a fresh copy of the image file each time.
- FUSE mounting is not allowed on certain NFS directories. Because of the way NFS permissions work, you can mount a FUSE file system on certain directories but not unmount it again, leaving zombie processes every time your program crashes. Misc.c checks for these directories and refuses to run in that case. The test scripts use a directory in /tmp, a non-NFS file system, to avoid this problem.

Debugging

Error - "`Block devices not permitted on fs`" - when unmounting file system:

You may see this error when trying to unmount a FUSE file system mounted on top of an NFS file system (i.e. your home directory). You shouldn't ignore it, because if you don't successfully unmount the file system you will leave running processes dangling, potentially causing problems.

This error occurs because the unmount command, *fusermount*, runs with root privileges on the local machine, but that the root user does not have privileges on NFS-mounted file systems. To fix this you need to make sure that all users have *execute* permission for all directories in the path; this error typically comes about because you are working in *~/classes*, which by default has permissions rwx------, forbidding access to all users except for the owner. To fix it, either do your work in another directory that is world-accessible, or grant permission to traverse (but not list) *~/classes* to all users:

```
chmod o+x ~/classes
```

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/
```

My implementation

As an example of how to go about this problem set, here is a vague description of some of the helper functions in my implementation:

- initialization – read the superblock and FAT into memory.
- parse path – split into array of strings. (if you allocate memory here, be sure to free it before you return from the FUSE method)
- lookup – takes a parsed path and finds the corresponding directory entry
- FAT operations – allocate a free block, free a block. These operate on an in-memory copy, then I call a flush function to write changes to disk.
- convert cs5600 attributes (dirent) to Linux (struct stat) – factored from readdir and getattr
- create_ent (factored common code from create and mkdir)

Blkdev interface

The block device abstraction we use is implemented in the following structure:

```
struct blkdev {
    struct blkdev_ops *ops;
    void *private;
};

#define BLOCK_SIZE 512   /* 512-byte unit for all blkdev addresses */

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

This is a common style of operating system structure, which provides the equivalent of a C++ abstract class by using a structure of function pointers for the virtual method table and a void* pointer for any subclass-specific data. Interfaces like this are used so that independently compiled drivers (e.g. network and graphics drivers) to be loaded into the kernel in an OS such as Windows or Linux and then invoked by direct function calls from within the OS.

The methods provided in the blkdev_ops structure are:

- `num_blks` - the total size of this block device, in 512-byte blocks
- `read` - read one or more blocks into a buffer. The caller guarantees that '`buf`' points to a buffer large enough to hold the amount of data being requested, and that `num_blks>0`. Legal return values are `SUCCESS` and `E_BADADDR`.

- `write` - write one or more blocks. The caller guarantees that 'buf' points to a buffer holding the amount of data being written, and that `num_blks>0`. Legal return values are `SUCCESS` and `E_BADADDR`.
- `close` - the destructor method, this closes the blkdev and frees any memory allocated.

The `E_BADADDR` error is returned if any address in the requested range is illegal - i.e. less than zero or greater than `blkdev->ops->num_blks(blkdev)`.

We will be working with disk image files, rather than actual devices, for ease of running and debugging your code. You may be familiar with image files in the form of .ISO files, which are byte-for-byte copies of a CD-ROM or DVD, and can be read by the same file system code which interacts with a physical disk; in our case we will be writing to the files as well.