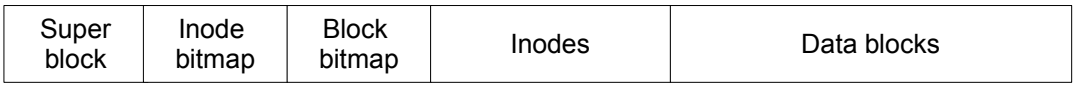
For this homework we are going to implement the fs7600 file system, a simple derivative of the Unix FFS file system. We will use the FUSE toolkit in Linux to implement the file system as a user-space process; instead of a physical disk we will use a data file accessed through a block device interface 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.

fs7600 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 structure can be used to implement the superblock:

```
struct fs7600_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

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

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.

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 mkfs-x6) 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 macros found in <select.h> can be used to access these bitmaps:

```
fd_set *map;
map = malloc(super.block_map_sz * FS_BLOCK_SIZE);
  … read bitmap from disk into 'map' …
if (FD_ISSET(100, map))     /* is entry 100 set? */
   FD_CLR(100, map);        /* clear it */
FD_SET(100, map);           /* set it again */
```

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 this homework we will use the C interface to the FUSE toolkit to create a program which can read, write, and mount CS7600fs 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 cs7600fs.

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:

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 method 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' and '-s' flags) The file system interface, in turn, allows a fs7600 image to be mounted as a standard Linux file system. The two interfaces are shown below:

```
pjd@bubbles$ ./homework 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$ ./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$
```

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.

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

You shouldn't see this error. If you do it's because you're trying to unmount a FUSE file system mounted on top of an NFS file system (i.e. your CCIS home directory when you're using a lab machine). Contact me if you have this error.

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.te 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 1024   /* 1024-byte unit for all blkdev addresses */

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

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 1024-byte blocks. (you won't need this, as the superblock tells you how big the file system is.)
- `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`.
- `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`.

These functions always return success. If you try to read or write an invalid address – i.e. less than zero or greater than the size of the device – an assert will fail in order to help you debug your code.

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.