Systems and Networks	**Project 2: User-Level File System**
CS3600 Fall 2014	**July 28, 2014**

This project is due at 11:59:59pm on October 27, 2014 and is worth 15% of your grade. You must complete it with a partner. You may only complete it alone or in a group of three if you have the instructor's explicit permission to do so for this project.

Note that there are **two** *milestone deadlines for this project, at 11:59:59pm on October 6, 2014, and at 11:59:59pm on October 16, 2014. More details are in the Milestone sections below.*

1 Description

Your job in this project is to design and implement a basic user-level file system. You will be given a block interface to a raw "disk," and, using this interface, you will be required to design a simple yet functional file system.

You will be using FUSE to implement this user level file system. In brief, FUSE allows you to hook up your file system to the Linux kernel, and your filesystem will appear just like any other. As a result, you can use any existing commands and tools to test your filesystem.

FUSE is already installed on most of the CCIS Linux machines, and you should be able to use those machines to develop, compile, and test your code. You are welcome to use your own machine to do the project. Of course, you need to install FUSE on your machine; it's available for Linux as well as FreeBSD. If your Linux Kernel is newer than 2.6.14, FUSE is already installed in your machine. However, you will need the FUSE headers (libfuse-dev for those of you on Ubuntu).

2 Requirements

Your file system must support a single-level directory structure, must support size, metadata, and permissions attributes, and must support create, delete, read, write, and move operations. You must also provide code which will format a new "blank" disk for your filesystem.

2.1 Simplifying assumptions

For the purposes of this project, you do not need to worry about the buffer cache; the OS will maintain the buffer cache for you and you don't need to do anything extra in your file system. You can assume all disk image reads and writes are synchronous (the OS actually caches data blocks and delays writes to disk, but for the purposes of this project let's ignore this detail). You do, however, need to maintain file metadata information somewhere, including the owner and group, permissions (mode), file size, and created/last accessed/last modified times. Finally, the block size is fixed to be 512 bytes (statically defined in `BLOCKSIZE`).

Your file system must support **at least 100 files**, allow **arbitrary-sized files** (up to the size of the total space available for storage), support **filenames of at least 27 characters**, support **the basic file metadata mentioned above**, and support **a single level of directory (i.e., "/")** (multiple directories can be supported for extra credit; see the section below).

2.2 Starter code

Very basic starter code for the assignment is available in `/course/cs3600f14/code/project2`. You must use this code as a basis for your project. Provided is a `3600fs` code and header file, and a `Makefile`. The `Makefile` is configured to correctly compile your code and to test your code on sample input. Also included is the outline of a `3600mkfs` program that you will write that will create a newly formatted disk (as a file).

To get started, you should copy down this directory into your own local directory (i.e., `cp -r /course/cs3600f14/code/project2 ~/cs3600`). You can compile your code by running `make`. You can also delete any compiled code and object files by running `make clean`.

To create a disk, you will run the `3600mkfs` command, with an argument of the number of disk blocks that will be in the disk:

```
$ ./3600mkfs 1000
```

You'll now see a `MYDISK` file in your local directory. Note that, right now the `3600mkfs` code doesn't actually do anything other than create the file. You'll need to fill out `3600mkfs` to have it create a disk using the format that you wish to use.

Once you've created your disk, you need to run the command to mount the file system. To run your code, you have to first create an empty directory, known as a *mount point*. *You should place this mount point in /tmp*. Then, you run `3600fs` with the arguments `-s -d` followed by that directory:

```
$ mkdir /tmp/fuse-dir
$ ./3600fs -s -d /tmp/fuse-dir
```

At this point, your user-level file system is now mounted. Now you can use the disk another shell:

```
$ cd /tmp/fuse-dir
$ ls
```

To unmount, you can either kill your original process, or you can run the command

```
$ fusermount -u /tmp/fuse-dir
```

2.3 Virtual disk

We are not mounting a physical disk; instead, we'll be using a fixed-size file on the normal file system. You will "mount" your filesystem by reading the disk image file provided. This disk image is simply a flat file which will eventually contain all the metadata blocks and data blocks in your file system. As a result, there is no need to worry about the physical characteristics of a disk (seek time, latency, disk layout). Instead, assume disk blocks are arranged in a linear order—i.e., block 3 is next to block 4. When testing your project we will, however, use the number of disk accesses and the distance between subsequent block numbers as a rough metric of your file system's performance.

2.3.1 Formatting the virtual disk

Before mounting a file system for the first time, you need to initialize the raw disk (in our case disk file). You will write another program that writes the super block information onto the disk file before using this file as a raw disk (essentially you are "formatting" the disk partition to be used as a file system). This file is `3600mkfs.c` in your tarball. What you write into the super block depends on how you design your file system, generally you would like to write some information about file system such as where the location of root directory or other metadata. You should initialize the root directory to be owned by the running user's UID and GID (e.g., `getuid()`, `getgid()`), and with mode 0777.

2.3.2 Raw disk interface

You will be provided with the following interface to the raw mounted disk from which you will construct and access your file system. Of course, you should not touch any of these functions.

`int dconnect()`
return value: 0 if successful, other values if an error occurred

`dconnect()` will connect to the disk (think of this like plugging in the USB drive containing your disk). You should call this function in only one page: `vfs_mount`.

`int dread(int blocknum, char *buf)`
blocknum: The block number you wish to read
buf: A pointer to at least `BLOCKSIZE` bytes of memory
return value: `BLOCKSIZE` if successful, negative values if an error occurred

Given a `blocknum`, `dread()` will read that block into the specified buffer `buf`. More specifically, `dread` will read `BLOCKSIZE` (bytes of data from the address logical block number `blocknum`. Thus, reads from the disk image only happen at the granularity of a block, simulating a real disk. Also, `dread()` is synchronous and will wait until the entire block is read and copied into `buf` before returning. Note that your code does not need to handle disk failure; it may simply exit with an error message (e.g., "your disk image crashed").

`int dwrite(int blocknum, const char *buf)`
blocknum: The block number you wish to write to
buf: A pointer to `BLOCKSIZE` bytes of memory
return value: `BLOCKSIZE` if successful, negative values if an error occurred

Given a block number, `dwrite()` will write a block from `buf` to the disk image at location `blocknum`. `dwrite()` is guaranteed to be atomic and synchronous, so there is no need to worry about cases where blocks themselves are partially written. When implementing, be sure to keep in mind that at any time the file system might crash so be sure that at all times you are in an recoverable state. However you are guaranteed that crashes can only occur before or after a `dwrite()` call and not during. Note that your code does not need to handle disk failure; it may simply exit with an error message (e.g., "your disk image crashed").

`int dunconnect()`
return value: 0 if successful, other values if an error occurred

`dconnect()` will disconnect the disk (think of this like unplugging in the USB drive containing your disk). You should call this function in only one page: `vfs_umount`.

2.4 File system API

The following is the API you are required to implement using the disk interface described above. When designing your file system, keep in mind that "crashes" may occur on any I/O operation and your filesystem must always be in a stable and consistent state. We will simulate crashes by modifying `dwrite()` to occasionally stop the execution of your program using `exit(1)` either before or after they complete. We will then ask your code to re-mount the disk image in the state it was left in at the time of the "crash."

The 3600`fs.c` file defines the following interface:

`void* vfs_mount(struct fuse_conn_info *conn)`

conn: You can ignore this argument
return value: void

This function expects the filename of he disk file that you are using as a disk; you may hard code the filename (defined in `3600fs.h` as `DISKFILE`). This function should mount your application-level filesystem stored in the file; once a mount occurs, that file will be used as the disk for all further file system commands you will implement (until an `vfs_unmount` occurs). Basically, you first need to connect the disk by calling `dconnect()`; this call is already provided for you. You will then need to check the integrity of your file system (start with the VCB and go from there). For example, if a crash occurred during a file create after an inode was allocated but before a directory entry is updated, such an error should be found and fixed in `vfs_mount`.

`void vfs_unmount (void *private_data)`

private_data: You can ignore this argument
return value: void

Unmount is responsible for unmounting the file system. You will need to write out any necessary meta data that might be required the next time the file system is mounted. For instance, it might note that the filesystem was cleanly unmounted, speeding up the integrity check the next time the file system is mounted. When you have written out all of your metadata and the disk is in a "clean" state, you should disconnect the disk by calling `dunconnect()`; this call is already provided for you.

`int vfs_getattr(const char *path, struct stat *stbuf)`

path: The path of the file being asked about (e.g., "/bar")
stbuf: Structure where the results are stored (see `man 2 stat`)
return value: 0 if file/directory found, `-ENOENT` if not found, -1 if an error occurred

Given an absolute path to a file or directory (i.e., "/foo"—all paths will start with the root directory of your file system, "/"), you need to return the file attributes that is similar stat system call. For this project, you don't need to fill in any attributes except file size, file size in blocks, the various metadata items (timestamps, mode, owner, group), and block size of the file system. `3600fs.c` has more information on this. Depending on your implementation of the file system, the specifics of the `vfs_getattr` might vary, but the basic steps your file system takes to get file attributes will be the following:

1. Resolve the file – this will involve getting the location of the entry (or location of data block with the file's inode) of the specified file.
2. Fill the file size, file size in blocks, metadata items, and block size information (note the block size is constant across all files).
3. You should return 0 on success or -1 on error. If the path doesn't exist, you should return `-ENOENT` (-2).

Note that you will have to set the mode to include the *type* of file system object the path refers to (regular file or directory). So, if the mode of the file is stored in `mode` (e.g., `mode = 0777`), you would do:

```
stbuf->st_mode = (mode & 0x0000ffff) | S_IFREG;
```

for a regular file and

```
stbuf->st_mode = (mode & 0x0000ffff) | S_IFDIR;
```

for a directory (or subdirectory).

`int vfs_create(const char *path, mode_t mode, struct fuse_file_info * fi)`

path: The path of the file being created (e.g., "/bar")
mode: The initial mode of the file (e.g., 0644)
fi: You can ignore this argument
return value: 0 if successful, `-EEXIST` if the file already exists, or -1 if another error occurred

Given an absolute path to a file (for example "/myFile"), `vfs_create` will create a new file named "myFile" in the "/" directory). The initial mode of the file should be `mode`, and the initial user ID and group ID can be set to be the values returned by `geteuid()` and `getegid()`, respectively. The create time, accessed time, and modified time should all be set to the current time; this can be retrieved by doing something like

```
struct timespec mytime;
clock_gettime(CLOCK_REALTIME, &mytime);
```

The steps your implementation may take to create a new file will likely include the following:

1. Make sure the file does not already exist
2. Allocate and initialize a new file entry
3. Update the directory by adding an entry for the newly created file.

You should return 0 on success or -1 on error (e.g., the path contains multiple directories, or file already exists).

`int vfs_read(const char *path, char *buf, size_t size, off_t offset, struct fuse_file_info *fi)`

path: The path of the file being read (e.g., "/bar")
buf: A pointer to the location where you should write the data
size: The maximum number of bytes you should read
offset: The location in the file (number of bytes from the beginning) to start reading
fi: You can ignore this argument
return value: The number of bytes read if successful, or -1 if an error occurred

The function `vfs_read` provides the ability to read data from an absolute path `path`, which should specify an existing file. It will attempt to read `size` bytes at offset `offset` from the specified file on your filesystem into the memory address `buf`. The return value is the amount of bytes actually read; if the file is smaller than size, `vfs_read` will simply return the most amount of bytes it could read. Note that you should only read less that `size` bytes if you've hit the end-of-file. On error, `vfs_read` will return -1. The actual implementation of `vfs_read` is dependent on how files are allocated.

`int vfs_write(const char *path, const char *buf, size_t size, off_t offset, struct fuse_file_info *fi)`

path: The path of the file being written to (e.g., "/bar")
buf: A pointer to the location where you should read the data from
size: The number of bytes you should write
offset: The location in the file (number of bytes from the beginning) to start writing
fi: You can ignore this argument
return value: The number of bytes written if successful, -`ENOSPC` if you run out of disk space, or -1 if another error occurred

The function `vfs_write` will attempt to write `size` bytes from memory address `buf` into a file specified by an absolute `path` (i.e., "/foo"—all paths will start with the root directory, "/") at offset `offset`. If the offset is greater than the file length, the file should be extended (with zeros) to be `offset` long before appending `buf` to the end. On error (e.g., the path does not exist) `vfs_write` will return -1, otherwise `vfs_write` should return the number of bytes written. Note size is not necessarily an integral number of blocks. Similar to `vfs_read`, the actual implementation of `vfs_write` will depend on how you decide to allocate file space.

`int vfs_truncate(const char *path, off_t offset)`

path: The path of the file being truncated (e.g., "/bar")
offset: The location in the file (number of bytes from the beginning) to truncate at
return value: 0 if successful, -1 if an error occurred

The function `vfs_truncate` will truncate the file specified by an absolute `path` at the offset `offset`. Thus, this call may end up freeing up disk blocks; if this is the case, these disk blocks should be made available for other files to use them (e.g., if you're using a FAT, they should be marked as free and added to the end of the free block list). On error (e.g., the path does not exist, or the offset is larger than the file) `vfs_truncate` will return -1, otherwise `vfs_truncate` should return 0.

`int vfs_delete(const char *path)`

path: The path of the file being deleted
return value: 0 if successful, -1 if an error occurred

This function deletes the specified filename (e.g., "/a"). Depending on your implementation of the file system, the specifics of the delete might vary, but the basic steps your file system takes to delete a file will be the following:

1. Make sure the file exists and is a file (not a directory)
2. Remove the file's entry from the directory
3. Free any data blocks used by the file

Again, these steps are very general and the actual logistics of how to locate data blocks of a file, how to free data blocks and how to update directory entries are dependent on your implementation of directories and file allocation. On error (e.g., the path does not exist) `vfs_delete` will return -1, otherwise `vfs_delete` should return 0.

```
int vfs_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t offset,
    struct fuse_file_info *fi)
```

path: The path of the directory being read (e.g., "/")
but: The buffer where the results will be placed
filler: A FUSE-provided function for putting together results
offset: The first directory entry to start reading
fi: You can ignore this argument
return value: 0 if successful, -1 if an error occurred

Given an absolute path to a directory, `vfs_readdir` will return all the files and directories in that directory. The steps you will take will be the following:

1. Make sure that `path` is "/" (otherwise, return -1)
2. Find the first directory entry following the given `offset`
3. Call the `filler` function with arguments of `buf`, the null-terminated filename, `NULL`, and the offset of the *next* directory entry
4. If filler returns nonzero, or if there are no more files, return 0
5. Otherwise, find the next file in the directory and go to step 3

You should return 0 on success or -1 on error (e.g., the directory does not exist). Note that, to support the `vfs_readdir` function, you will need to use the FUSE-provided `filler()` function. More details can be found online (e.g., `http://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/unclea`

```
int vfs_rename(const char *from, const char *to)
```

from: The path of the file/directory being renamed
to: The new name
return value: 0 if successful, -1 if an error occurred

The function `vfs_rename` will rename a file named by the string `from` and rename it to the file name specified by `to`. If the file `to` already exists, that file should first be deleted before the `from` file is renamed. As usual, return 0 on success, -1 on failure.

`int vfs_chmod(const char *file, mode_t mode)`
file: The path of the file/directory being changed
to: The new name
return value: 0 if successful, -1 if an error occurred

The function `vfs_chmod` will change the permissions mode of the file to be `mode`. As usual, return 0 on success, -1 on failure.

`int vfs_chown(const char *file, uid_t uid, gid_t gid)`
file: The path of the file/directory being changed
uid: The new user who owns the file
gid: The new group who owns the file
return value: 0 if successful, -1 if an error occurred

The function `vfs_chown` will change the user ID of the file to be `uid` and the group ID of the file to be `gid`. As usual, return 0 on success, -1 on failure.

`int vfs_utimens(const char *file, const struct timespec ts[2])`
file: The path of the file/directory being changed
ts: An array containing the new access time and the new modification time
return value: 0 if successful, -1 if an error occurred

The function `vfs_utimens` will update both the last access time of the file (to be `ts[0]`) and the last modification time of the file (to be `ts[1]`). As usual, return 0 on success, -1 on failure.

3 Extra credit (30 points)

You can extend your file system to support recursive directories (i.e., directories other than just "/"). All of the above API calls must work with your filesystem, and you must additionally support the `vfs_mkdir` API call:

`int vfs_mkdir(const char *path, mode_t mode)`
path: The path of the directory being created
mode: The initial mode (permission) of the directory
return value: 0 if successful, -1 if an error occurred

The function `vfs_mkdir` creates a directory at the specified path, with the initial mode of `mode`. The function is not expected to make multiple directories (i.e., if only `/foo` exists and a request is received to create `/foo/bar/baz`, you should return an error). As usual, return 0 on success, -1 on failure.

4 Implementation hints

In this project you are free in designing the details of the implementation of your file system. However, it is highly recommended that you model your file system after an existing implementation, as this way you can be sure your design is safe and you will also have a means of reference during implementation. The following are two methods of file allocation that are recommended. The first is the File Allocation Table (FAT), and the second is the Unix File System inode. You should be able to find more information about these file systems in the course textbook. Additionally, please refer to the Project 2 Implementation Notes document for details on the suggested method of implementation.

You should develop your client program on the CCIS Linux machines, as these have the necessary compiler and library support. You are welcome to use your own Linux/OS X machines, but you are responsible for getting your code working, and your code *must* work when graded on the CCIS Linux machines. If you do not have a CCIS account, you should get one ASAP in order to complete the project. Your code must be `-Wall` clean on `gcc`. Do not ask the TA for help on (or post to the forum) code that is not `-Wall` clean unless getting rid of the warning is what the problem is in the first place.

For this project, if you wish to write debugging messages to the console, you must write to `stderr`. You can do this using code like

```
fprintf(stderr, "your message %d\n", here);
```

You will want to make sure that you break your program up into modules, such that each module represents a sensible type abstraction. If you have questions on how to do this, please come see me or the TAs.

5 Suggested implementation strategy

Very roughly, once you've decided on your file system layout, you should get `vfs_mount` and `vfs_umount` working (at first, these functions will do almost nothing other than read your VCB). Then, implement `vfs_getattr`, being sure to return the correct type of object (`S_IFDIR`/`S_IFREG`) and returning -`ENOENT` if the file doesn't exist. At this point, you should be able to run `ls` on your mounted file system.

Next, implement `vfs_readdir`, `vfs_create`, `vfs_delete`, and `vfs_rename`. You should then be able to create and delete files, and should be able to pass the second milestone. To do this, you should probably implement helper functions for finding a file, finding a blank location for a new file, and so forth.

Next, implement `vfs_read` and `vfs_write`. You'll likely want to implement helper functions for finding the *n*th block of a file, for pushing and popping blocks on and off the free block list (or, if you're not using such a list, functions for finding a free block and marking a block as free). Finally, implement the metadata functions like `vfs_chown`. Only attempt the extra credit once you have a fully working single-level file system.

In this project, there are "pairs" of functions that both need to implemented correctly before you can test either. For example, `vfs_read` and `vfs_write`, or `vfs_readdir` and `vfs_create`. Keep this in mind during your implementation; you can potentially implement parts of each function and then test them together (e.g., support very small reads/writes, then move to bigger reads/writes).

6 Testing

First test that your filesystem by formatting a disk, creating a local directory, and mounting the disk to that directory. All UNIX/Linux programs will see the directory as any other directory, and should be able to read and write files just like any other file system. Thus, tools like `cat`, `ls`, `rm`, `touch`, etc should all work with your filesystem.

In general, you should use the temporary directories of CCIS Linux machines as the location to mount your filesystem. If you mount your filesystem on an NFS-mounted directory (e.g., somewhere in your home directory), this will often lead to stale mount points that will not disappear. In more detail, you should run something like

```
mkdir /tmp/foo-amislove
./3600fs -s -d /tmp/foo-amislove
```

which will cause your filesystem to be mounted in `/tmp/foo-amislove`.

Additionally, we have included a basic test script to check the output of your filesystem against our reference solution and check your code's compatibility with the grading script. If your code fails in the test script we provide, you can be assured that it will fare poorly when run under the grading script. To run the test script, simply type

```
bash$ make test
```

This will compile your code and then test your file system on a number of inputs, comparing the results against the reference solution. If any errors are detected, the test will print out the expected and actual output. For example, you might see something like

```
bash$ make test
./test
...
  Create file                                   FAIL
    No file existed
  Create second file                            FAIL
    No file existed
  Create third file                             FAIL
    No file existed
...
bash$
```

We include a few sample tests, but these are by no means exhaustive. We expect that you will create additional test to ensure that your file system behaves as expected.

7 Submitting your project

7.1 Registering your team

You and your partner should first register as a team by running the `/course/cs3600f14/bin/register` script. You should pick out a team name (no spaces or non-alphanumeric characters, please) and run

```
/course/cs3600f14/bin/register project2 <teamname>
```

This will either report back success or will give you an error message. If you have trouble registering, please contact the course staff.

You must register your team by 11:59:59pm on October 1, 2014.

7.2 Milestone 1

In order to ensure that you are making sufficient progress, you will have an interim milestone deadline. For the milestone, you must submit a draft of your README, which describes the disk layout that you've chosen and why you've made this choice. Your `3600mkfs` must be able to make a blank disk with your layout format, and your `3600fs` code must support the `vfs_mount` API calls. In other words, your code must allow us to create a blank disk and mount it; you do not have to support creating and deleting files or reading and writing file data.

You should submit your milestone by running the `/course/cs3600f14/bin/turnin` script. Specifically, you should create a `project2` directory, and place all of your code in it. Then, run

```
/course/cs3600f14/bin/turnin project2-milestone1 <dir>
```

Where `<dir>` is the name of the directory with your submission. The script will print out every file that you are submitting, make sure that it prints out all of the files you wish to submit! Additionally, you should receive a confirmation number once your submission is successfully submitted; if you do not receive a confirmation number, your submission was not received.

You must submit your milestone by 11:59:59pm on October 6, 2014. No slip days can be used on milestone 1.

7.3 Milestone 2

In order to make sure that you are making sufficient progress on your implementation, there will be a second milestone. Your `3600fs` code must support the `vfs_mount`, `vfs_create`, `vfs_delete`, `vfs_readdir`, and `vfs_getattr` API calls. Similarly, your `3600mkfs` code should create a disk that your `3600fs` code can support. In other words, your code must allow us to create a disk, mount it, and then create and delete files and list the contents of the directory; you do not have to support reading and writing file data or handle most file metadata. This translates roughly to being able to create and read FCBs.

You should submit your milestone by running the `/course/cs3600f14/bin/turnin` script. Specifically, you should create a `project2` directory, and place all of your code in it. Then, run

```
/course/cs3600f14/bin/turnin project2-milestone2 <dir>
```

Where `<dir>` is the name of the directory with your submission. The script will print out every file that you are submitting, make sure that it prints out all of the files you wish to submit! Additionally, you should receive a confirmation number once your submission is successfully submitted; if you do not receive a confirmation number, your submission was not received.

You must submit your milestone by 11:59:59pm on October 16, 2014. No slip days can be used on milestone 2.

7.4 Final submission

For the final submission, you should submit you (thoroughly documented) code along with a plain-text (no Word or PDF) README file. In this file, you should describe your high-level approach, the challenges you faced, a list of properties/features of your design that you think is good, and an overview of how you tested your code. You should also describe the disk layout that you chose, and why you made that choice.

You should submit your project by running the `/course/cs3600f14/bin/turnin` script. Specifically, you should create a `project2` directory, and place all of your code and README files in it. Then, run

```
/course/cs3600f14/bin/turnin project2 <dir>
```

Where `<dir>` is the name of the directory with your submission. Again, the script will print out every file that you are submitting, make sure that it prints out all of the files you wish to submit! Additionally, you should receive a confirmation number once your submission is successfully submitted; if you do not receive a confirmation number, your submission was not received.

You must submit your project by 11:59:59pm on October 27, 2014.

8 Grading

The grading in this project will consist of

60% Program functionality

10% Performance

15% Style and documentation

5% Milestone 1 functionality

10% Milestone 2 functionality

You are, however, going to be graded on how efficiently you use the disk. In other words, how large a file can be stored on your disk? What is the usable capacity of your file system if it's full of many 1-byte files? How many disk blocks do you need to read to access a small file? You should consider these factors when thinking about how to design your file system.

9 Advice

A few pointers that you may find useful while working on this project:

- You should test your filesystem with standard UNIX utilities, such as ls, rm, touch, cat, echo, stat, dd.
- Check the Piazza forum for question and clarifications. You should post project-specific questions there first, before emailing the course staff.

- Finally, get started early and come to the TA lab hours; these are held in the lab at 102 West Village H. You are welcome to come to the lab and work, and ask the TA any questions you may have.