CS OS: Assignment 3
Simple File System

0. Assignment Statement

In order to get a more complete view of how filesystems interact with disks and block devices, you'll build a simple one yourself. Rather than sending commands to the device driver to operate the disk, you'll instead use FUSE, a library that allows you to implement a Filesystem in USErspace.

1. Background

Filesystems provide the 'file' abstraction. Standing between the user and the disk, they organize data and indexing information on the disk so that metadata like 'filename', 'size', and 'permissions' can be mapped to a series of disk blocks that correspond to a 'file'. Filesystems also handle the movement of data in to and out of the series of disk blocks that represent a 'file' in order to support the abstractions of 'reading' and 'writing' from/to a 'file'. You will be implementing a fairly simple file system using FUSE.

FUSE is a kernel module that redirects system calls to your filesystem from the OS to code you write at user level. While working with an actual disk through a device driver is worthy endeavor, we would prefer you concentrate more on your filesystem's logic and organization instead of building one while also learning how to operate a disk directly. You will however have to emulate working with a disk by storing all data in a flat file that you will access as a block device. All user data as well as all indexing blocks, management information or metadata about files must be stored in that single flat file using whatever organization or format you wish. All your FUSE calls must only reference this "virtual disk" and will provide the 'file' abstraction by fetching segments from the "disk" in a logical manner. For instance, if a call is made to read from a given file descriptor, your library will get the request, will have to look up the index information to find out where the 'file' is located in your "disk", determine which block would correspond to the offset indicated by the file handle, read in that disk block, and write the requested segment of the data block to the pointer given to you by the user. To give your file system implementation some context and scope, stub files will be provided.

2.1. Assignment Requirements

As described above, you will implement a simple file system using FUSE and a flat file. Be sure you use no more than 16MB and can index at least 128 files.

2.2. Minimal Requirements

Your scheduler and memory manager **do not** have to coexist with your filesystem. Your filesystem may be disjunct from the other two.

You should implement the following system calls at minimum:

0. int create(char* path)
1. int delete(char* path)
2. int stat(char* path, struct stat* buf)
3. int open(char* path)
4. int close(int fileID)
5. int read(int fileID, void* buffer, int bytes)
6. int write(int fileID, void* buffer, int bytes)
7. struct dirent* readdir(int directoryID)

Since you will be using FUSE, you will implement these system calls by filling in code for the appropriate FUSE functions. Beyond the system calls, you will also need to implement some FUSE functions like init() and destroy() that allow FUSE to set up and operate your file system. See section 3.2 for a list of which FUSE functions cover these system calls and perform other useful operations. Your file data block and indexing/information blocks can take any structure you like, however you should use an indexing mechanism that holds a series of pointers to direct-mapped blocks. A filesystem that uses direct-mapped indexes is required. You may want to look at how the original UNIX filesystem (UFS) is implemented, and especially examine the structure of i-nodes.

2.3. Extensions and Improvements

You must implement at least one of the improvements below, but will receive additional credit for each improvement after the first:

Additional Indirection Only direct-mapped file indexes are required, however in order to store larger files indexed file systems usually use multiple indexing layers. You will receive additional credit for implementing single-, and then again for double-indirection in your file system.

Directory Support We only require you to implement file interactions and readdir to allow the filesystem to be mounted. If you want to provide directory support to lay the groundwork for your filesystem to use directories, you could implement the informational directory functions for additional credit:

0. int opendir(char* path)
1. int closedir(int directoryID)

Extended Directory Operations Once you have the informational directory operations above, you can now build functions to create and remove directories and you can freely traverse your filesystem, for additional credit:

(a) int mkdir(char* path)
(b) int rmdir(char* path)

3. Recommended Procedure

- Download, build and test the stub code, as detailed in subsection 4.1 below.
- Read through the *.c and *.h files in the assignment2/src directory of the stub code
- Make a copy of the stub code, write some simple test code to execute filesystem operations on the example filesystem and look at the log to see how and with what parameters the functions were called.
- First write some simple code to test out the functions in block.c to emulate reading and writing information in 'disk blocks' to your flat file.
- Design your file system's architecture in order to store file metadata and index data blocks.
- Start up the the stub code to get a simple file system (SFS) that outputs logging information, and run some filesystem commands in it to determine which FUSE functions are used by each system call.

For instance:
$ ../src/sfs testfsfile mountdir
sfs init()
sfs opendir(path=”/”, fi=0xe8b29c50)
sfs getattr(path=”/.xdg-volume-info”, statbuf=0xe37fdc20)
sfs readdir(path=”/”, buf=0xe4000ca0, filler=0xe931b530, offset=0, fi=0xe37fdc50)
sfs releasedir(path=”/”, fi=0xe8b29c60)

$ ls
getattr(path=”/”, statbuf=0x7eed1bf0)

touch ./mountdir/bbb.b
getattr(path=”/bbb.b”, statbuf=0x7e6d0c20)
create(path=”/bbb.b”, mode=0100664, fi=0x7eed1d10)
getattr(path=”/bbb.b”, statbuf=0x7e6d0b60)
release(path=”/bbb.b”, fi=0x7decfd10)

$ echo ”bb” >> ./mountdir/bbb.b
getattr(path=”/bbb.b”, statbuf=0x7e6d0c20)
open(path”/bbb.b”, fi=0x7eed1d10)
write(path=”/bbb.b”, buf=0x7faaf060, size=3, offset=3, fi=0x7e6d0d80)
release(path=”/bbb.b”, fi=0x7decfd10)

$ cat ./mountdir/bbb.b
getattr(path=”/bbb.b”, statbuf=0x7e6d0c20)
open(path”/bbb.b”, fi=0x7eed1d10)
read(path=”/bbb.b”, buf=0x74000af0, size=4096, offset=0, fi=0x7decfd10)
release(path=”/bbb.b”, fi=0x7decfd10)

$ rm ./mountdir/bbb.b
getattr(path=”/bbb.b”, statbuf=0x7e6d0c20)
unlink(path=”/bbb.b”)

You will probably want to implement your system calls in the following order and granularity:
0. init, stat (getattr), readdir
1. create, delete (unlink)
2. open, release
3. write
4. read

This way you can test your implementation of your new commands using your previous ones, so you can build as you go.

3.1. Stub Code
We have provided you with a basic set of stub code that will give you a framework to build your filesystem within.

Build Stub Code:
- download assignment3-stub.tar.gz
- tar xvf assignment3-stub.tar.gz
- cd assignment3
- ./configure
- make

Build Test Infrastructure:
- mkdir /.freespace/<NetID>
- touch /.freespace/<NetID>/testfsfile
- mkdir /tmp/<NetID>
- mkdir /tmp/<NetID>/mountdir

Run Stub Code:
- cd src
- ./sfs /.freespace/<NetID>/testfsfile /tmp/mountdir
- Check if the filesystem was mounted successfully: mount | grep sfs

Didn't work:

Did work:
sfs on /.autofs/tmp/<NetID>/mountdir type fuse.sfs (rw,nosuid,nodev,user= username)

- To unmount the filesystem: fusermount -u /tmp/deymious/mountdir

This basic filesystem doesn't do a whole lot. It mostly writes out log messages whenever you use a filesystem command on it. Take a look at sfs.c to see these basic implementations of the filesystem commands. Around line 289 you can see the fuse operations struct, where the common filesystem commands are aliased to FUSE function pointers. Besides the filesystem commands there are a few FUSE commands you should implement.

3.2 Basic FUSE elements
All the FUSE commands you will need to write to implement the basic filesystem commands and operate your file system are outlined below.

• .init = sfs init
Initialize the filesystem. This function can often be left unimplemented, but it can be a handy way to perform one-time setup such as allocating variable-sized data structures or initializing a new filesystem. The fuse conn info structure gives information about what features are supported by FUSE, and can be used to request certain capabilities. The return value of this function is available to all file operations in the private data field of fuse context. It is also passed as a parameter to the destroy() method. See additional information on init below in subsection 4.1.

• .destroy = sfs destroy
Called when the filesystem exits. The private data comes from the return value of init.

• .create = sfs create
Create and open a file.

• .unlink = sfs unlink
(the FUSE name for delete())
Remove (delete) the given file, symbolic link, hard link, or special node. Note that if you support hard links, unlink only deletes the data when the last hard link is removed. For our purposes, presume that you will only be working with files.

• .getattr = sfs getattr
(the FUSE name for stat())
Return file attributes. The ”stat” structure is described in detail in the stat(2) manual page. For the given pathname, this should fill in the elements of the ”stat” structure. If a field is meaningless or semi-meaningless (e.g., st ino) then it should be set to 0 or given a ”reasonable” value. This call is pretty much required for a usable filesystem.

• .open = sfs open
Open a file. If you aren’t using file handles, this function should just check for existence and permissions and return either success or an error code. If you use file handles, you should also allocate any necessary structures and set struct fuse file info -¿fh. In addition, struct fuse file info has some other fields that an advanced filesystem might find useful; see the structure definition for addition information.

• .release = sfs release, (the FUSE name for close())
This is the only FUSE function that doesn’t have a directly corresponding system call, although close(2) is related. Release is called when FUSE is completely done with a file; at that point, you can free up any temporarily allocated data structures. The IBM document claims that there is exactly one release per open, but I don’t know if that is true.

• .read = sfs read
Read bytes from the given file into the buffer, beginning offset bytes into the file. See read(2) for full details. Returns the number of bytes transferred, or 0 if offset was at or beyond the end of the file. Required for any sensible filesystem.

• .write = sfs write
Write bytes from the given buffer into a file. See write(2) for full details. Returns the number of bytes transferred, or 0 if offset was at or beyond the end of the file. Required for any sensible filesystem.

• .readdir = sfs readdir
Return one or more directory entries (struct dirent) to the caller. This is one of the most complex FUSE functions. It is related to, but not identical to, the readdir(2) and getdents(2) system calls, and the readdir(3) library function. Required for essentially any filesystem, since it's what makes ls and a whole bunch of other things work. See subsection 4.2 below for additional information on readdir.

3.3 Extended FUSE elements
If you go on to implement the additional directory support elements, you will need to implement the following.

• .opendir = sfs opendir
Open a directory for reading.

• .releasedir = sfs releasedir
This is like release, except for directories.

• .rmdir = sfs rmdir
Remove the given directory. This should succeed only if the directory is empty (except for "." and "..").
See rmdir(2) for details.

• .mkdir = sfs mkdir
Create a directory with the given name. The directory permissions are encoded in mode. See mkdir(2) for details. This function is needed for any reasonable read/write filesystem.

3.4 Submission
After you have finished your filesystem, compress your directory into assignment3SFS.tar.gz and submit it on Sakai. You should also submit a PDF describing the design of your filesystem, including the physical organization of your indexes and data structures in the flat file and how they are used to implement the logical operations required by the filesystem.

Additional Function Detail

4.1 Init detail

The initialization function accepts a fuse conn info structure, which can be used to investigate and control the system's capabilities. The components of this structure are:

• proto major and proto minor

Major and minor versions of the FUSE protocol (read-only).

• async read

On entry, this is nonzero if asynchronous reads are supported. The initialization function can modify this as desired. Note that this field is duplicated by the FUSE CAP ASYNC READ flag; asynchronous reads are controlled by the logical OR of the field and the flag. (Yes, this is a silly hangover from the past.)

• max write

The maximum size of a write buffer. This can be modified by the init function. If it is set to less than 4096, it is increased to that value.

• max readahead

The maximum readahead size. This can be modified by the init function.

• capable

The capabilities supported by the FUSE kernel module, encoded as bit flags (read-only).

• want

The capabilities desired by the FUSE client, encoded as bit flags.

The capabilities that can be requested are:

• FUSE CAP ASYNC READ

Use asynchronous reads (default). To disable this option, the client must clear both this capability (in the want flags) and the async read field. If synchronous reads are chosen, Fuse will wait for reads to complete before issuing any other requests.

• FUSE CAP POSIX LOCKS

Set if the client supports ”remote” locking via the lock call.

• FUSE CAP ATOMIC O TRUNC

Set if the filesystem supports the O TRUNC open flag.

• FUSE CAP EXPORT SUPPORT

Set if the client handles lookups of ”.” and ”..” itself. Otherwise, FUSE traps these and handles them.

• FUSE CAP BIG WRITES

Set if the filesystem can handle writes larger than 4 KB.

• FUSE CAP DONT MASK

Set to prevent the umask from being applied to files on create operations. (Note: as far as I can tell from examining the code, this flag isn't actually implemented.)

4.2 Readdir detail

The readdir function is somewhat like read, in that it starts at a given offset and returns results in a caller-supplied buffer. However, the offset not a byte offset, and the results are a series of struct dirents rather than being uninterpreted bytes. To make life easier, FUSE provides a ”filler” function that will help you put things into the buffer.

The general plan for a complete and correct readdir is:

0. Find the first directory entry following the given offset (see below).
1. Optionally, create a struct stat that describes the file as for getattr (but FUSE only looks at st ino and the file-type bits of st mode).
2. Call the filler function with arguments of buf, the null-terminated filename, the address of your struct stat (or NULL if you have none), and the offset of the next directory entry.
3. If filler returns nonzero, or if there are no more files, return 0.
4. Find the next file in the directory.
5. Go back to step 2.

From FUSE’s point of view, the offset is an uninterpreted off t (i.e., an unsigned integer). You provide an offset when you call filler, and it’s possible that such an offset might come back to you as an argument later. Typically, it’s simply the byte offset (within your directory layout) of the directory entry, but it’s really up to you. It’s also important to note that readdir can return errors in a number of instances; in particular it can return EBADF if the file handle is invalid, or -ENOENT if you use the path argument and the path doesn’t exist.