ECE 454/750T10, Spring 2014 — Assignment 4
Due Fri, July 25, 11:59:59 PM

(Follow the submission instructions at the end. If you are in 454, mention the names of both group members in the README.)

Objective: realize a simplified distributed file system.

You need to realize an infrastructure for a simplified distributed file system. Your infrastructure comprises a client-side API and a server. The client-side API allows a client application to mount, browse and edit files and folders from the server-side. The server communicates with the client to serve files and folders.

Server

Your server is a program that takes one command-line argument. That argument is the name of a local folder which is to be served to clients. Following is a sample run of your server program.

```
% ./fsServer ../myfolder
129.97.56.12 10002
```

As indicated above, your server takes as command-line input the name of a local folder whose contents it serves. It prints out the public IP address/domain name and port at which it can be reached.

You can assume that *myfolder* contains regular files and subfolders only. You serve only *myfolder* and any subfolders, and not its parent or any other folders.

Client-side API

For the client side, you are to provide an API that a client-side application can use to communicate with your server. Your client-side API must comprise the following functions. We provide a file `ece454_fs.h` that has `extern` declarations for these functions.

- *int fsMount(const char *srvIpOrDomName, const unsigned int srvPort, const char *localFolderName)* — used by the client application to mount the remote server folder locally, and have it be referred to as *localFolderName*. Returns 0 on success, and -1 on failure. The global variable `errno` is set appropriately on failure.
- *int fsUnmount(const char *localFolderName)* — the counterpart of *fsMount()* to unmount a remote filesystem that is referred to locally by *localFolderName*. Returns 0 on success, -1 on failure with `errno` set appropriately.
- *FSDIR* fsOpenDir(const char *folderName)* — opens the folder *folderName* that is presumably the local name of a folder that has been mounted previously with *fsMount()*. Note that this could be a subfolder within the folder that has been mounted. The return type is a pointer to *FSDIR*, which you define. It should be NULL in case of error, and `errno` should be set appropriately.
- *int fsCloseDir(FSDIR *)* — the counterpart of *fsOpenDir()*. After this call, the *FSDIR* * argument is no longer valid. Returns 0 on success, and -1 on failure; `errno` is set on failure.
- *struct fsDirent *fsReadDir(FSDIR *)* — returns the next entry in the folder to which the argument refers. That is, a caller can repeatedly call *fsReadDir()* to "browse" a folder for its contents. The return type is a pointer to something of type *struct fsDirent*, which is is defined in `ece454_fs.h` as follows:

```
struct fsDirent {
    char entName[256]; /* Name of entry */
    unsigned char entType; /* 0 for file, 1 for folder,
                              -1 otherwise. */
}
```

The return is NULL is either an error occurs, or if we reach the end of the folder's contents. In case of error, `errno` is set appropriately. In the latter case, `errno` is set to 0 so that the caller knows that no error occurred.

- *int fsOpen(const char *fname, int mode)* — opens a file whose path is *fname*. The mode is one of two values: 0 for read, and 1 for write. Returns a file descriptor that can be used in future calls for operations on this file (see below). If the file does not exist and mode is write (i.e., 1), then the file should be created. The permissions up on creation should be that clients are able to read and write the file. "Write" means that the file will be overwritten by subsequent *fsWrite()* calls. The return is positive if no error occurs. Otherwise, it is -1, and `errno` is set appropriately.
- *int fsClose(int)* — the close counterpart of *fsOpen()*. The argument file descriptor is no longer valid after this call. Returns 0 on success, -1 on failure; `errno` set on failure.
- *int fsRead(int fd, void *buf, const unsigned int count)* — used to read up to *count* bytes in to the supplied buffer *buf* from the file referred to by the file descriptor *fd*, which was presumably the return from a call to *fsOpen()* in read-mode. This call should not read past the end of the file. Subsequent *fsRead()* calls should return subsequent bytes in the file, i.e., the same behaviour as read(2). The return is the number of bytes actually read and filled into *buf*. The return is -1 on error, `errno` set appropriately.
- *int fsWrite(int fd, const void *buf, const unsigned int count)* — writes up to *count* bytes from *buf* into the file referred to with the file descriptor *fd* that was presumably the return from an earlier *fsOpen()* call in write-mode. If the file already exists, the first call to *fsWrite()* with a newly opened file descriptor overwrites the file with the contents of *buf*. Subsequent calls to *fsWrite()* to the same open file descriptor should append. (This is the same behaviour as for write(2) for regular files.) Returns the number of bytes actually written; -1 on error, with `errno` set appropriately.
- *int fsRemove(const char *)* — removes (i.e., deletes) this file or folder from the server. Of course, this can cause concurrency issues.

Transport

You can use your simplified RPC implementation from the earlier programming assignment as the underlying layer. You can also use mine, which I will provide on Learn soon. Mine will implement the exact same API as your earlier programming assignment. But mine will use TCP and not UDP, and therefore may be more reliable. You can also reimplement the transport from scratch, and not use anything from the earlier assignment.

Memory management

The space for *FSDIR* that is returned by *fsOpenDir()* and is an argument to *fsCloseDir()* and *fsDirEnt()* is yours. That is, you allocate space for it. Note that we may make multiple calls to *fsOpenDir()* before any call to *fsCloseDir()*. So you could allocate space dynamically in *fsOpenDir()*, and deallocate that space in *fsCloseDir()*.

The space for *struct fsDirent* to which a pointer is returned by *fsReadDir()* is yours. You can choose to allocate this statically. That is, a caller remains warned that another call to *fsReadDir()* could overwrite the contents of an earlier call.

All other memory to which the calls refer belongs to the client application.

Concurrency

You need to somehow manage concurrent client accesses. That is, multiple clients should be allowed to *fsMount()* the same server, and those clients must be able to concurrently issue calls. No client should get an error on account on concurrent access. We will test this. For example, we may have two clients c_1, c_2, where c_1 bombards your server with accesses to a file, and c_2 infrequently accesses the file, but c_2 gets to do the final write on the file, well after c_1 has ceased operations.

Makefile

You must include a Makefile with your submission. The Makefile should have three targets: clean, server and client-api.a. Issuing 'make clean' should remove all executable, `.o` and `.a` files from your folder and all subfolders. Issuing 'make server' should build your server application. Issuing 'make client-api.a' should result in the file `libclient-api.a` that has the client API that our client application can link to. (See the sample unpacking script.)

Coding Standard

Your code must follow a programming style that you can choose freely; but you have to explicitly specify which one in comments in your code towards the start. It is important that the code is easy to read, with meaningful variables names. Example coding conventions are at `https://code.google.com/p/google-styleguide/`.

Submission Instructions

- Your submission is made to the appropriate dropbox on Learn.
- It must be a single zip file. You can use the "`-r`" option to `zip` to recursively zip a folder.
- The name of your submission must be: `ece454a4.zip`. Our script will look for this, and if it does not find it for you, you get an automatic 0 (see marking scheme below).
- Unzipping your `ece454a4.zip` must result in a single folder called `ece454a4`. You are allowed to have whatever subfolders you want, but your Makefile should be at the top-level folder, i.e., in the folder `ece454a4`.
- Include a README file with the names of your two group-members if you are in 454. If you are in 750T10, then of course you need to do this assignment on your own. Include your name only in the README.

Evaluation

Our grading scheme:

a) Marking script breaks, e.g., because submission not structured as per instructions, makefile does not work, program crashes, etc. — automatic 0.

b) 5% if you implemented something meaningful that compiles and runs.

c) 30% if you implemented something meaningful that compiles and runs, and we are able to write simple client applications, and everything works for a simple folder structure at the server.

d) 90% everything works for our more complex tests, e.g., more complex folder structure, multiple simultaneous mounts, etc.

e) 100% if, in addition to meeting the functional requirements, your code is pretty and well-documented.