Project 4 -- Network File Server

Worth: 15 points
Assigned: March 24, 2023
Due: April 17, 2023

1. Overview

In this project, you will implement a multi-threaded network file server. Clients that use your file server will interact with it via network messages. This project will help you understand hierarchical file systems, socket programming, client-server systems, and network protocols, and it will give you experience building a substantial multi-threaded program with fine-grained locking.

Your file server will provide a hierarchical file system. Files and directories stored on the file server are referred to by a full pathname, with `/` being the delimiting character. For example, the pathname `/class/482/notes` refers to a file `notes` that is stored in the directory `/class/482`. Pathnames must start with `/`, and they must not end with `/`.

Directories store files and/or sub-directories; files store data. Each file and directory is owned by a particular user, except for the root directory `/`, which is owned by all users. Users may only access files and directories they own. Note that while '/' is a directory, a client cannot name it directly, because its name ('/') is not legal.

2. Client interface to the file server

After initializing the library (by calling `fs_clientinit`), a client uses the following functions to issue requests to your file server: `fs_readblock`, `fs_writeblock`, `fs_create`, and `fs_delete`. These functions are described in `fs_client.h` and included in `libfs_client.o`. Each client program should include `fs_client.h` and link with `libfs_client.o`.

Here is an example client that uses these functions. Assume the file system is initially empty. This client is run with two arguments:

1. the name of the file server's computer
2. the port on which the file server process is accepting connections from clients.

```
#include <iostream>
#include <cassert>
#include <cstdlib>
#include "fs_client.h"

using std::cout;

int main(int argc, char *argv[]) {
    char *server;
    int server_port;

    const char *writedata = "We hold these truths to be self-evident, that all men ar

    char readdata[FS_BLOCKSIZE];
    int status;

    if (argc != 3) {
        cout << "error: usage: " << argv[0] << " <server> <serverPort>\n";
        exit(1);
    }
    server = argv[1];
    server_port = atoi(argv[2]);

    fs_clientinit(server, server_port);

    status = fs_create("user1", "/dir", 'd');
    assert(!status);

    status = fs_create("user1", "/dir/file", 'f');
    assert(!status);

    status = fs_writeblock("user1", "/dir/file", 0, writedata);
    assert(!status);

    status = fs_readblock("user1", "/dir/file", 0, readdata);
    assert(!status);

    status = fs_delete("user1", "/dir/file");
    assert(!status);

    status = fs_delete("user1", "/dir");
    assert(!status);
}
```

3. Communication protocol between client and file server

This section describes the request and response messages used to communicate between clients and the file server. The client's side of this protocol is carried out by the functions in `libfs_client.o`. You will write code in your file server to carry out the file server's side of the protocol.

There are four types of requests that can be sent over the network from a client to the file server: `FS_READBLOCK`, `FS_WRITEBLOCK`, `FS_CREATE`, `FS_DELETE`. Each client request causes the client library to open a connection to the server, send the request, receive the response from the server, and close its side of the connection.

After responding to a client's request, the server should close its side of the connection. If the file server receives a client request that causes an error, it should close its side of the connection **without sending a response message**, then continue processing other requests.

Sections 3.1-3.4 describe the format of each client request message and the server's response message. Correct requests will follow the specified format exactly, e.g., no extra spaces, all fields non-empty, numbers in canonical base-10 form with no leading zeroes.

Since clients are untrusted, your file server should be careful in how it handles network input (processing requests from untrusted sources is an important aspect of writing operating systems). Your file server must continue correct operation regardless of what input it receives from clients. Avoid making assumptions about the content and size of the requests until you have verified those assumptions.

While your file server must handle client requests correctly, the specified communication protocol does not guarantee that requests and responses are authentic, confidential, or fresh, nor does it permit file servers to defend against denial-of-service attacks (e.g., clients that open connections but never send complete requests). The given protocol could be layered on top of a secure communication protocol (e.g., TLS) to help provide these guarantees, but this is outside the scope of our class.

3.1 FS_READBLOCK

A client reads a block of an existing file by sending an `FS_READBLOCK` request to the file server.

An `FS_READBLOCK` request message is a C string of the following format:

```
FS_READBLOCK <username> <pathname> <block><NULL>
```

- `<username>` is the name of the user making the request
- `<pathname>` is the name of the file being read
- `<block>` specifies which block of the file to read

- `<NULL>` is the ASCII character '\0' (terminating the string)

Upon receiving an `FS_READBLOCK` request, the file server should check if the request is valid. If so, the file server should read the requested data from disk and return the data in the response message. The response message for a successful `FS_READBLOCK` is the same as the request message, followed by the data that was read from the file (note that this data is outside of the response C string).

3.2 FS_WRITEBLOCK

A client writes a block to an existing file by sending an `FS_WRITEBLOCK` request to the file server.

An `FS_WRITEBLOCK` request message is a C string (plus the data to be written) of the following format:

```
FS_WRITEBLOCK <username> <pathname> <block><NULL><data>
```

- `<username>` is the name of the user making the request
- `<pathname>` is the name of the file to which the data is being written
- `<block>` specifies which block of the file to write. <block> may refer to an existing block in the file, or it may refer to the block immediately after the current end of the file (this is how files grow in size).
- `<NULL>` is the ASCII character '\0' (terminating the string)
- `<data>` is the data to write to the file. Note that `<data>` is outside of the request string (i.e., after `<NULL>`).

Upon receiving an `FS_WRITEBLOCK` request, the file server should check if the request is valid and there is space on the disk and in the file. If so, the file server should write the data to the file and respond to the client. The response message for a successful `FS_WRITEBLOCK` is the C string portion of the request message (i.e., the request message without the data being written to the file).

3.3 FS_CREATE

A client creates a new file or directory by sending an `FS_CREATE` request to the file server.

An `FS_CREATE` request message is a C string of the following format:

```
FS_CREATE <username> <pathname> <type><NULL>
```

- `<username>` is the name of the user making the request

- `<pathname>` is the name of the file or directory being created
- `<type>` can be 'f' (file) or 'd' (directory)
- `<NULL>` is the ASCII character '\0' (terminating the string)

Upon receiving an `FS_CREATE` request, the file server should check if the request is valid and there is space on the disk and in the directory. If so, the file server should create the new file or directory. The response message for a successful `FS_CREATE` is the same as the request message.

3.4 `FS_DELETE`

A client deletes an existing file or empty directory by sending an `FS_DELETE` request to the file server.

An `FS_DELETE` request message is a C string of the following format:

```
FS_DELETE <username> <pathname><NULL>
```

- `<username>` is the name of the user making the request
- `<pathname>` is the name of the file or directory being deleted
- `<NULL>` is the ASCII character '\0' (terminating the string)

Upon receiving an `FS_DELETE` request, the file server should check if the request is valid. A directory can only be deleted when it contains no files or sub-directories. The root directory `/` cannot be deleted. If the request is valid, the file server should delete the file or directory.

The response message for a successful `FS_DELETE` is the same as the request message.

4. File system structure on disk

This section describes the file system structure on disk that your file server will read and write. `fs_param.h` (which is included automatically in both `fs_client.h` and `fs_server.h`) defines the basic file system parameters.

`fs_server.h` has two definitions that describe the on-disk data structures:

```
/*
 * Definitions for on-disk data structures.
 */
struct {
        char name[FS_MAXFILENAME + 1];         // name of this file or directory
        uint32_t inode_block;                  // disk block that stores the inode
                                               // for this file or directory (0 if
                                               // this direntry is unused)
    } fs_direntry;

    struct {
        char type;                             // file ('f') or directory ('d')
        char owner[FS_MAXUSERNAME + 1];        // owner of this file or directory
        uint32_t size;                         // size of this file or directory
                                               // in blocks
        uint32_t blocks[FS_MAXFILEBLOCKS];     // array of data blocks for this
                                               // file or directory
    } fs_inode;
```

Each file and directory is described by an inode, which is stored in a single disk block. The structure of an inode is specified in `fs_inode`. The `type` field specifies whether the inode refers to a file (type f) or directory (type d). The `owner` field is the name of the user that created the file or directory. It is a string of characters (whitespace characters are not allowed) that is terminated by a null character.

The root directory `/` is owned by all users, and its `owner` field is the empty string. All other files and directories may be read, written, and deleted only by their owner. The `blocks` array lists the disk blocks where this file or directory's data is stored. Entries in the `blocks` array that are beyond the end of the file may have arbitrary values. The inode for the root directory `/` is stored in disk block 0.

The data for a directory is an array of `fs_direntry` entries (one entry per file or sub-directory). Unused directory entries are identified by `inode_block=0`; this is safe because block 0 is known to be the root directory, and cannot be anything else. In an array of directory entries, entries that are used may be interspersed with entries that are unused, e.g., entries 0, 5, and 15 might be used, with the rest of the entries being unused. At least one entry should be used in each block of directory entries.

Each directory entry contains the name of a file or directory (including the '\0' that terminates the string) and the disk block number of the disk block that stores that file or directory's inode. A file or directory name is a non-empty string of characters (whitespace and `/` characters are not allowed).

Each disk block may be used by at most one file or directory.

Hint: the definitions above serve two purposes. The first purpose is to concisely describe the data format on disk. E.g., an `fs_direntry` consists of `FS_MAXFILENAME+1` bytes for the file name, followed by a 4-byte unsigned integer (in little-endian byte order on x86 systems). The second purpose is to provide an easy way to convert the raw data you read from disk into a data structure, viz. through typecasting.

5. File server internals

This section discusses and guides some design choices you will encounter when writing the file server. Your file server should include the header file `fs_server.h` .

5.1 Arguments and input

Your file server should be able to be called with 0 or 1 command-line arguments. The argument, if present, specifies the port number the file server should use to listen for incoming connections from clients. If there is no argument, the file server should have the operating system choose a port. For example, your file server could be started either as:

```
fs 8000
```

or as:

```
fs
```

5.2 Initialization

When your file server starts, it should carry out the following tasks:

- Initialize the list of free disk blocks by reading the relevant data from the existing file system. Your file server should be able to start with any valid file system (an empty file system as well as file systems containing files).
- Set up the socket that clients will use to connect to the file server, including calling `listen` (Section 8).

After these initialization steps are complete, your file server should print the port number of the socket that clients will use to connect to the file server (regardless of whether it was specified on the command line or chosen by the system). Here's the statement to use (substitute `port_number` with your own variable):

```
std::cout << "\n@@@ port " << port_number << std::endl;
```

5.3 Concurrency and threads

The workload to your file server may include any number of concurrent client requests. Your file server should create a thread for each request, so it can service requests from an arbitrary number of client threads at the same time. Use the C++ Standard Library concurrency mechanisms for this (e.g. `std::thread`), not the native `pthread` mechanism. After you create a thread, you should call `std::thread::detach` so its resources are freed when the thread finishes. The main thread in your file server (which is created automatically when your process starts) should not exit -- if the main thread exits, the autograder will think the file server has exited.

One goal of this project is to service concurrent client requests whenever it is safe to do so. A thread that is executing a blocking system call (receiving data from the network and reading or writing the disk) to service one request should not block another thread, unless required for safety.

Use a lock (`std::mutex`) to protect your in-memory data structures. Use the basic interface for `std::mutex` (`lock()`, `unlock()`), or better, use an RAII wrapper (construct `std::lock_guard` or `std::unique_lock`).

Use locks (`std::mutex`) or (for **advanced** projects) reader/writer locks (`std::shared_mutex`) to protect disk data. Each lock or reader/writer lock in your program should cover exactly one file or directory. You need not (and should not) assign locks on a smaller granularity.

Core:

- The core version of the file server should use locks (`std::mutex`) to protect disk data. Locks allow each file or directory to be accessed (read or written) by only one thread at a time. For example, `FS_READBLOCK` and `FS_WRITEBLOCK` accesses a file, and `FS_CREATE` and `FS_DELETE` accesses the directory that holds the file/directory being created/deleted. Traversing the file system requires the server to safely access each of the directories along the path being traversed.
- The core version may use reader/writer locks (see below), but this is not required.

Advanced:

- The advanced version of the file server must use reader/writer locks (`std::shared_mutex`) to protect disk data. Use the low-level interface for `std::shared_mutex` (`lock()`, `unlock()`, `lock_shared()`, `unlock_shared()`), or better, use an RAII wrapper (construct `std::unique_lock` or `std::shared_lock`).
- Think about when a file or directory is read or written, and ensure that a file or directory can be written by only one thread at a time, while still allowing it to be read by multiple threads (if no thread is writing it). For example, `FS_READBLOCK` reads a file, `FS_WRITEBLOCK` writes a file, and `FS_CREATE` and `FS_DELETE` write the directory that holds the file/directory being

created/deleted. Traversing the file system requires the server to read each of the directories along the path being traversed.

- Minimize the space needed for locks. In particular, your file server should not always require more memory for locks as the size of the disk grows or the number of files/directories in the file system increases.

Use **hand-over-hand locking** when traversing a chain of file system entities: lock the first entity, access the first entity as needed, lock the second entity, then release the lock on the first entity, and so on. Think about what race condition is possible if you release the lock on the first entity before locking the second entity. **Hint:** Thinking about pathname traversal as a recursive process helps simplify one's locking design, even if it is implemented iteratively.

When deleting a file or directory, you may need to hold multiple locks. Think about what race condition is possible if you only lock the directory that holds the file or directory being deleted.

Verify that your file server executes requests in parallel if and only if it is safe to do so. First, issue one request and cause the thread servicing that request to block indefinitely (e.g., by calling sleep just before doing a slow operation). Then issue a second request that should not be serviced concurrently with the first, but do not call sleep on the second request. Verify that the second request blocks on the first request. Do the same test with two requests that should be able to be serviced concurrently, and verify that the second thread completes. As you conduct these tests, keep track of what you've tested in a spreadsheet (first request; second request; which slow operation you blocked in the first request; test case name). Note that there are many possible pairs of requests and there are several slow operations you could block for each request, so this should be a long table.

Hint: One way to accomplish this is to use "special" file/directory names to indicate that the file server should pause for several seconds the first time it performs a particular action on that file/directory. Then, create a non-empty file system with the right name(s) so that you can test this more easily. **Hint:** This testing can be tedious! You can reduce the need to do it by thinking carefully about internal abstractions and using RAII to manage locks. Time spent in architectural design will be repaid many-fold in reduced time spent chasing bugs. **Hint:** RAII uses **static scope** to manage lock lifetimes, but hand-over-hand locking follows a **dynamic execution** pattern. You can get around this apparent conflict by using C++ move semantics.

The autograder limits the stack size per thread to 1 MB (the default value of 10 MB is too large to support highly concurrent workloads). You can test with the same limit by using `ulimit` command in bash.

5.4 Performance and caching

Your file server should receive network data efficiently when it is reasonable to do so, and should minimize the number of disk I/Os used to carry out requests. Most file servers cache disk information in memory aggressively to reduce disk I/Os. However, to simplify the project, your file server should **not** cache information between requests. All work for a request must be performed **before** responding to the client. The only information about disk state that your file server should cache in memory between requests is the list of free disk blocks. **Hint:** it is reasonable to receive delimited messages of unknown length byte-by-byte, but the same is not true for large blocks of data where the size is known in advance. You do not need to worry about clients that leave a connection open, but never send some expected data. Such denial-of-service attacks are beyond the scope of this project.

5.5 Managing and reading directory entries

Directory data consists of an array of `fs_direntry` entries and is stored in an array of disk blocks. The size of a directory is always an integer number of blocks, so many directories will have unused directory entries (identified by `inode_block=0`).

When `FS_CREATE` allocates a directory entry, it should choose the lowest-numbered directory entry that is unused.

When `FS_DELETE` deletes a file, the directory entry for that file is marked unused. Usually, `FS_DELETE` should not move directory entries around to compact the directory data; it should simply leave existing entries in place. The exception to this is when an `FS_DELETE` leaves *all* directory entries in a disk block unused. In this case, `FS_DELETE` should shrink the directory by an entire disk block by updating the directory inode. To do so, `FS_DELETE` should remove the unused block from the directory inode's `blocks` array then shift all the following values in the `blocks` array up by one.

The file server will need to read directory data to carry out most client requests. It should read directory data in the order of the `blocks` array in the directory inode.

5.6 File system consistency and order of disk writes

Your file server must maintain a consistent file system on disk, regardless of when the system might crash. This implies a specific ordering of disk writes for file system operations that involve multiple disk writes. The general rule for file systems is that meta-data (e.g., directory entry or inode) should never point to anything invalid (e.g., invalid inode or data). Thus, when writing a block of data and a block containing a pointer to the data block, one should write the block being pointed to before writing the block containing the pointer.

E.g., for `FS_CREATE`, the file server should write the new inode to disk before writing the directory block (which points to that inode). If the file server mistakenly wrote out the directory block before the inode block, a crash in between these two writes would leave the directory pointing at a garbage inode block. In the same way, you should reason through the order of disk writes for `FS_WRITEBLOCK` and `FS_DELETE` so that the file system remains consistent regardless of when a crash occurs.

5.7 Parsing

Parsing untrusted input can be tricky. Here are some suggestions for how to do this more easily.

Use the C++ facilities meant for parsing strings; do not recreate these facilities. The format of (correct) network requests can be processed easily by istringstream (e.g., operator>>). You may also use regular expressions; they are more powerful than needed for this project, but useful to learn.

Since client input is untrusted, parsing requests via istringstream (or any other method) might fail. If your parsing is correct for correct requests, then any parsing failures indicate an incorrect request.

Finally, remember that your parsing functionality must be safe. It's ok for istringstream to fail, but it's not ok for your file server to fail.

6. Utility functions and utility programs

Your file server must use the utility functions `disk_readblock` , and `disk_writeblock` to access the disk. These functions are declared in `fs_server.h` and are included in `libfs_server.o` . Use the standard functions `send` , `recv` , and `close` to send network messages, receive them, and close network sockets.

Use `disk_readblock` and `disk_writeblock` to read and write a disk block. You may assume these functions are atomic with respect to crashes (but not necessarily with respect to each other; use locking to avoid conflicts). Minimize the number of disk I/Os. These functions access the disk data stored in the Linux file `/tmp/fs_tmp.<uniqname>.disk` , where `<uniqname>` is the login ID of the person running the file server.

You can run the utility program `showfs` to show the current file system contents stored in the simulated disk (which is stored in the host file `/tmp/fs_tmp.<uniqname>.disk`). Remember to set the execute permission bit on `showfs` (e.g., run " `chmod +x showfs` ").

You can initialize a file system in the simulated disk by running the utility program `createfs` . Remember to set the execute permission bit on `createfs` (e.g., run " `chmod +x createfs` "). To create an empty file system, run `createfs` without any arguments. To create a non-empty file system, run " `createfs image.fs` ", where `image.fs` contains the output (or edited output) generated by `showfs` .

Use `send` and `recv` to send and receive network messages. Remember that the `len` parameter to `send` and `recv` specifies the *maximum* amount of data to send or receive; these functions may send or receive less than this. Your file server should send a response back to the client only after all processing for that request is finished.

Use `close` to close network sockets.

7. Output

Your file server must produce the output mentioned in Section 5.2. The infrastructure will also produce output for calls to `disk_readblock` , `disk_writeblock` , `send` , and `close` . In addition, your file server may produce any output you need for debugging, as long as that output does not contain lines that start with `@@@` .

Because your file server is multi-threaded, you must be careful to prevent output from different threads from being interleaved. To prevent garbled output, your file server must declare:

```
extern std::mutex cout_lock;
```

and protect each use of `std::cout` using the `cout_lock` mutex (this mutex is also used by `libfs_server.o` when it produces output). **Note:** do not try to acquire any other lock while holding `cout_lock` , or deadlock is likely.

8. Sockets and TCP

A significant part of this project is learning how to use BSD sockets, which is a common programming interface used in network programming. Unfortunately, the socket interface is baroque. This section contains a little help for using sockets, but we expect you to get many necessary details by reading the relevant manual pages. Start with the `tcp` manual page. The class web page also contains a tutorial on how to use sockets and TCP.

Start by using the `socket` function to creating a socket, which is an endpoint for communication. The `tcp` manual page tells you how to create a socket for TCP. It's usually a good idea (though not strictly necessary) to configure the socket to allow it to reuse local addresses. Use `setsockopt` with level SOL_SOCKET and optname SO_REUSEADDR to configure the socket. This avoids the annoying `bind: address already in use` error that you would otherwise get when you kill the file server and restart it with the same port number.

After creating the socket, the next step is to assign a port number to the socket. This port number is what a client will use to connect to the file server. Use `bind` to assign a port number to a socket. **Note: do not include "using namespace std" in your source file, because `std::bind` conflicts with `bind()`**. Here's how to initialize the parameter passed to `bind`:

```
#include <cstring>
struct sockaddr_in addr;

memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(port_number);

bind(sock, (struct sockaddr*) &addr, sizeof(addr));
```

See the `ip(7)` manual page for an explanation of INADDR_ANY. `htonl` and `htons` are used to convert host integers and shorts into network format (network byte order) for use by `bind`. If `port_number` is 0 in the above code, `bind` will have the system select a port.

Use `getsockname` to get the port number assigned by the system. Use `ntohs` to convert from the network byte order returned by `bind` to a host number.

After binding, use `listen` to configure the socket to allow a queue of pending connection requests. A queue length of 30 is sufficient.

Use `accept` to accept a connection on the socket. `accept` creates a new socket that can be used for two-way communication between the two parties. A client process will use `connect` to initiate a connection to the file server.

Use `recv` to receive a network message, and `send` to send a network message.

Clients may shut down the connection before the file server has sent the response. Sending to a connection that is shut down generates a `SIGPIPE` signal, which would terminate the file server. To avoid this, use the `MSG_NOSIGNAL` flag when calling `send`.

Don't forget to close the socket (using `close`) after you're done servicing the request, otherwise you'll quickly run out of free file descriptors.

9. Test cases

An integral (and graded) part of writing your file server will be to write a suite of test cases to validate any file server. This is common practice in the real world--software companies maintain a suite of test cases for their programs and use this suite to check the program's correctness after a change. Writing a comprehensive suite of test cases will deepen your understanding of file systems, and it will help you debug your file server. To construct a good test suite, think about what different things might happen on each type of request (e.g., what effect different blocks might have on the disk I/Os needed to satisfy a request).

Each test case for the file server will be a short C++ client program that uses a file server via the interface described in Section 2 (e.g., the example program in Section 2). The name of each client program should start with `test` and end with `.cc` or `.cpp` , e.g., `test1.cpp` .

Each client program you submit may be accompanied by an optional file system image with the same base name; e.g., client program `test1.cpp` may be accompanied by file system image `test1.fs` . Client programs that are accompanied by a `.fs` file will be run on a file system initialized with that file system image; i.e., the autograder will run `createfs` on that image before starting the file server). Client programs that are not accompanied by a `.fs` file will be run on an empty file system.

The format of a `.fs` file matches the output from `showfs` , and can also be used as input to `createfs` .

Each test case should be run with exactly two arguments:

1. the hostname that is running the file server
2. the port that the file server is listening on for client connections

Test cases should use no other input, and should exit(0) when run with a correct file server. When we run your test cases, we will start the file server with the file system image you specify. Test cases without accompanying file system images will be started with an empty file system.

Your test suite may contain up to 20 test cases. Each test case may cause a correct file server to generate at most 6000 lines of `@@@` output and take less than 60 seconds to run. These limits are larger than needed to expose all buggy file servers (however, you will probably need to run larger test cases on your own to test your file server). You will submit your suite of test cases together with your file server, and we will grade your test suite according to how thoroughly it exercises a file server. See Section 11 for how your test suite will be graded.

You should test your file server with both serial and concurrent client requests. However, your submitted test suite need only be a single process issuing a single request at a time; none of the buggy file servers used to evaluate your test suite require multiple concurrent requests to be exposed. **Hint:** Many of our test cases test for behaviors you cannot submit as test cases to the autograder. It is **very common** for a group to catch enough instructor bugs for full test-case credit, but still not pass the majority of the autograder test cases for your file system.

10. Project logistics

Write your file server in C++17 on Linux. Our "ground truth" compilation toolchain is provided by CAEN. At the time of this writing, that is version 8.5.0.

You may use any part of the standard C++ library, including the C++ thread facilities. You should not use any libraries other than the standard C++ library. Your file server code may be in multiple files. Each file name must end with `.cc`, `.cpp`, or `.h` and must not start with `test`.

This Makefile shows how to compile a file server and an application that uses the file server (adjust the file names in the Makefile to match your own program).

You are **required** to document your development process by having your Makefile run `autotag.sh` each time it compiles your file server (see Makefile above). `autotag.sh` creates a git tag for a compilation, which helps the instructors better understand your development process. `autotag.sh` also configures your local git repo to include these tags when you run "`git push`". To use it, download `autotag.sh` and set its execute permission bit (run "`chmod +x autotag.sh`"). If you have several local git repos, be sure to push to github from the same repo in which you compiled your file server.

We have created a private github repository for your group (`eecs482/<group>.4`), where `<group>` is the sorted, dot-separated list of your group members' uniqnames. Initialize your local repository by cloning the (empty) repository from github, e.g.,

```
git clone git@github.com:eecs482/uniqnameA.uniqnameB.4
```

11. Grading, auto-grading and formatting

To help you validate your programs, your submissions will be graded automatically, and the results will be provided to you. You may then continue to work on the project and re-submit. The results from the autograder will not be very illuminating; they won't tell you where your problem is or give you the test programs. The main purpose of the autograder is to help you know to keep working on your project (rather than thinking it's perfect and ending up with a 0). The best way to debug your program is to generate your own test cases, figure out the correct answers, and compare your program's output to the correct answers. This is also one of the best ways to learn the concepts in the project.

Here is a (very rough) categorization of some of the test cases used by the autograder. Some test cases are too special-purpose to categorize; others appear in multiple categories.

- 0-10,25-27: basic functionality
- 11-17,26,63,65-67: error handling
- 18-23: large, serial (i.e., non-concurrent) test cases
- 26-58: start with pre-existing file systems
- 24,28-67: concurrent test cases

The student suite of test cases will be graded according to how thoroughly they test a file server. We will judge thoroughness of the test suite by how well it exposes potential bugs in a file server. The autograder will first run a test case with a correct file server to generate the right answers for this test case. The autograder will then run the test case with a set of buggy file servers. A test case exposes a buggy file server by causing the buggy file server to generate output (on `stdout`) that differs from correct file server's output or by causing the buggy file server to generate a file system image on disk (i.e., output from `showfs`) that differs from that generated by a correct file server. The test suite is graded based on how many of the buggy file servers were exposed by at least one test case. This is known as *mutation testing* in the research literature on automated testing.

You may submit your program as many times as you like, and all submissions will be graded and cataloged. We will use your highest-scoring submission, with ties broken in favor of the later submission. If any group member is in EECS 498-002, your highest-scoring submission will be chosen using a (2/3,1/3) weighted average of your core and advanced scores.

You must recompile and `git push` at least once between submissions.

The autograder will provide feedback for the first submission of each day, plus 3 bonus submissions over the duration of this project. Bonus submissions will be used automatically--any submission you make after the first one of that day will use one of your bonus submissions. After your 3 bonus submissions are used up, the system will continue to provide feedback for the first submission of each day.

Because your programs will be auto-graded, you must be careful to follow the exact rules in the project description:

- Your code must not print any output lines that start with `@@@`, except for the output specified in Section 5.2.
- Do not modify the header files provided in this handout.
- Your file server must use `disk_readblock` and `disk_writeblock` to write the disk, `send` to send messages, and `close` to close a network socket.

- The file server should send a response back only after all processing for that request is finished.
- When `FS_CREATE` allocates a directory entry, it should choose the lowest-numbered free directory entry.
- When `FS_DELETE` frees a directory entry, it should leave other entries in place, except when it can shrink the directory by an entire disk block by updating only the directory inode (in which case it should remove the unused block from the directory inode's blocks array, then shift all the following values in the blocks array up by one).
- Your file server should read directory data in the order of the blocks array in the directory's inode.

In addition to the autograder's evaluation of your program's correctness, a human grader will evaluate your program on issues such as documentation, coding style, the efficiency, brevity, and understandability of your code, compile warnings, etc.. Your documentation should explain the synchronization scheme followed by your file server. Your final score for each project part will be the product of the hand-graded score (between 1-1.04) and the autograder score.

12. Turning in the project

Submit the following files for your file server:

- C++ files for your file server. File names should end in `.cc`, `.cpp`, or `.h` and must not start with `test`. Do not submit the files provided in this handout.
- Suite of test cases. Each test case should be in a single file. File names should start with `test` and end with `.cc` or `.cpp`.

Each person should also describe the contributions of each team member using the following web form. Submissions after the due date will automatically use up your late days; if you have no late days left, late submissions will not be counted for you (though they may still count for other members of your group, if they have more late days available).

13. Files included in this handout (zip file)

- `createfs`
- `fs_client.h`
- `fs_param.h`
- `fs_server.h`
- `libfs_client.o`
- `libfs_server.o`

- `showfs`
- `autotag.sh`
- `Makefile`

14. Experimental platforms

The files provided in this handout were compiled on RHEL 8. They should work on most other Linux distributions (e.g., Ubuntu) and on Windows Subsystem for Linux (WSL), but these are not officially supported.

We also provide an experimental, unsupported version of the infrastructure for students who want to develop on MacOS. If you are developing on MacOS:

- Use `libfs_client_macos.o` instead of `libfs_client.o` .
- Use `libfs_server_macos.o` instead of `libfs_server.o` .
- Use `createfs_macos` instead of `createfs` .
- Use `showfs_macos` instead of `showfs` .
- Add `-D_XOPEN_SOURCE` to the compilation flags.