Project 4 -- network file server

Worth: 14 points
Assigned: Friday, March 24, 2017
Due: Tuesday, April 18, 2017

1. Overview

In this project, you will implement a multi-threaded, secure 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 security protocols.

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.

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_session`, `fs_readblock`, `fs_writeblock`, `fs_create`, `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 server was initialized with a user `user1` whose password is `password1`. 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 <cstdlib>
#include "fs_client.h"

using namespace std;

int main(int argc, char *argv[])

{
    char *server;
    int server_port;
    unsigned int session, seq=0;

    const char *writedata = "We hold these truths to be self-evident, that all men are created equal, that they are endowed by their Creator with c

    char readdata[FS_BLOCKSIZE];

    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);
    fs_session("user1", "password1", &session, seq++);
    fs_create("user1", "password1", session, seq++, "/dir", 'd');
    fs_create("user1", "password1", session, seq++, "/dir/file", 'f');
    fs_writeblock("user1", "password1", session, seq++, "/dir/file", 0, writedata);
    fs_readblock("user1", "password1", session, seq++, "/dir/file", 0, readdata);
    fs_delete("user1", "password1", session, seq++, "/dir/file");
    fs_delete("user1", "password1", session, seq++, "/dir");
}
```

3. Encryption

All request and response messages between the client and file server will use encryption (secret-key encryption based on the user's password). The file server will be given a list of user and passwords when it is started (see Section 6.1). `fs_param.h` (automatically included in `fs_client.h` and `fs_server.h`) defines the maximum size of a username and password as `FS_MAXUSERNAME` and `FS_MAXPASSWORD`.

We will provide encryption and decryption functions (described in Section 7).

Each request message from the client (described in Section 4) will be encrypted using the password parameter that was passed to the client function. To enable the file server to decrypt the request message, the client will send a cleartext (i.e., un-encrypted) request header before sending the request message. The cleartext request header is a null-terminated C string with the following format (note the space between `<username>` and `<size>`):

`<username> <size><NULL>`

- `<username>` is the name of the user that was passed to the client function. The file server uses this information to choose which password to use to decrypt the ensuing request message.
- `<size>` is the size of the encrypted message that follows this cleartext request header
- `<NULL>` is the ASCII character '\0' (terminating the string)

When the file server receives a request, it will decrypt the request using the information provided in the cleartext request header. If the file server can not decrypt the request (e.g., `<username>` is unknown or the client uses the wrong password), the file server handles this as it does other erroneous requests (i.e., by closing the connection without sending a response).

Each response message from the file server (described in Section 4) will be encrypted using the user's password. To enable the client to receive and decrypt the response message, the file server will send a cleartext response header before sending the response message. The cleartext response header is a null-terminated C string with the following format:

```
<size><NULL>
```

- `<size>` is the size of the encrypted message that follows this cleartext response header
- `<NULL>` is the ASCII character '\0' (terminating the string)

4. 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 five types of requests that can be sent over the network from a client to the file server: `FS_SESSION`, `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.

Section 4.1-4.5 describe the format of each client request message and the server's response message. Note the spacing in the message formats; these must be **exactly** as specified.

4.1 `FS_SESSION`

A client requests a new session with `FS_SESSION` (a user can call `FS_SESSION` any number of times). Client requests use a session and sequence number (both unsigned) as a unique identifier for the request (a nonce) to thwart replay attacks. A user may only use session numbers that have been returned by that user's prior `FS_SESSION` requests. A session remains active for the lifetime of the file server.

The first request in a session is the `FS_SESSION` that created the session, which uses the specified sequence number. Each subsequent sequence number for a session must be larger than all prior sequence numbers used by that session (they may increase by more than 1). The sequence number for a session gets "used" by any client request in that session, as long as the client request is made by the user that created that session. An erroneous request uses up its sequence number just like a correct request (think about what attack could be executed otherwise), as long as the file server can decrypt and parse the session/sequence numbers from the request and the session is owned by that user.

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

```
FS_SESSION <session> <sequence><NULL>
```

- `<session>` is 0. This field is unused for FS_SESSION requests. It's here just to make the formats of all request messages more uniform.
- `<sequence>` is the sequence number for this request
- `<NULL>` is the ASCII character '\0' (terminating the string)

Upon receiving an `FS_SESSION` request, the file server should assign a number for the new session, which should be the smallest number that has not yet been assigned to a session. Session numbers are global across all users, and the first returned session number should be 0. The file server should then send the following response message:

```
<session> <sequence><NULL>
```

- `<session>` is the new session number
- `<sequence>` is the sequence from the request message
- `<NULL>` is the ASCII character '\0' (terminating the string)

4.2 `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 <session> <sequence> <pathname> <block><NULL>
```

- `<session>` is the session number for this request
- `<sequence>` is the sequence number for this 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` follows the following format:

```
<session> <sequence><NULL><data>
```

- `<session>` is the session number from the request message
- `<sequence>` is the sequence from the request message
- `<NULL>` is the ASCII character '\0' (terminating the string)
- `<data>` is the data that was read from the file. Note that `<data>` is outside of the response string (i.e., after `<NULL>`).

4.3 `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 of the following format:

```
FS_WRITEBLOCK <session> <sequence> <pathname> <block><NULL><data>
```

- `<session>` is the session number for this request
- `<sequence>` is the sequence number for this 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` follows the following format:

```
<session> <sequence><NULL>
```

- `<session>` is the session number from the request message
- `<sequence>` is the sequence from the request message
- `<NULL>` is the ASCII character '\0' (terminating the string)

No data should be written to the file for unsuccessful requests.

4.4 `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 <session> <sequence> <pathname> <type><NULL>
```

- `<session>` is the session number for this request
- `<sequence>` is the sequence number for this 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` follows the following format:

```
<session> <sequence><NULL>
```

- `<session>` is the session number from the request message
- `<sequence>` is the sequence from the request message
- `<NULL>` is the ASCII character '\0' (terminating the string)

4.5 `FS_DELETE`

A client deletes an existing file or 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 <session> <sequence> <pathname><NULL>
```

- `<session>` is the session number for this request
- `<sequence>` is the sequence number for this 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, and 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` follows the following format:

```
<session> <sequence><NULL>
```

- `<session>` is the session number from the request message
- `<sequence>` is the sequence from the request message
- `<NULL>` is the ASCII character '\0' (terminating the string)

5. 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
} fs_direntry;

struct {
    char type;                          // file ('f') or directory ('d')
    char owner[FS_MAXUSERNAME + 1];
    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, including the '\0' that terminates the string. The root directory `/` is owned by all users, and its `owner` field is the empty string. 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`. 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. 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 `/` are not allowed).

Tip: 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.

6. 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`.

6.1 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 system choose a port.

Your file server will be passed a list of usernames and passwords via stdin (the file stream read by cin). Each line will contain

```
<username> <password>
```

For example, the Linux file `passwords` could contain the following contents:

```
user1 password1
user2 password2
user3 password3
user4 password4
```

and your file server could be started as:

```
fs 8000 < passwords
```

or

```
fs < passwords
```

You may assume that usernames and passwords passed to your file server are non-empty, are of legal length, and contain only letters and numbers.

6.2 Initialization

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

- Read the list of usernames and passwords from stdin.
- 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 9).

After these initialization steps are done, 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):

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

6.3 Concurrency and threads

The workload to your file server may include any number of concurrent client requests. Your file server should use multiple threads (via C++11 threads), so it can service requests from an arbitrary number of client threads at the same time.

Create a thread for each request and synchronize between these threads. 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. Doing so will make the auto-grader think the file server exited.

One goal of this project is to service multiple 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 other threads, unless required for safety.

Here are some steps to help you design your concurrency scheme.

1. Think about the main entities stored in the file system (files and directories) and when each entity is read or written. For example, `FS_READBLOCK` reads a file, and

`FS_WRITEBLOCK` writes a file. `FS_CREATE` and `FS_DELETE` write the directory that holds the file/directory being created/deleted. Traversing the file system to find a file or directory `x` requires the server to read each of the directories between `/` and `x`.
2. Use a locking scheme that allows each file system entity to be written by only one thread at a time, but allows it to be read by multiple threads (if no thread is writing it). Each lock in your scheme should cover exactly one file system entity (file or directory); you need not (and should not) assign locks on a smaller granularity. Lock each file system entity before accessing it.
3. Think carefully about when to release a lock on a file system entity. Releasing a lock too late will unnecessarily limit concurrency, while releasing it too early will create race conditions. 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. What race condition is possible if you release the lock on the first entity before locking the second entity?.
4. In addition to a lock per file-system entity, you will need a lock that protects your in-memory data structures.
5. Hint: when deleting a file or directory, you will need to hold two 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 two requests that should not be serviced concurrently and verify that one request blocks the other. For example, writing a directory (e.g., due to `FS_CREATE`) should prevent other requests from traversing that directory.

Second, issue two requests that should be able to be serviced concurrently and verify that they take place concurrently. For example, `FS_WRITEBLOCK` of one file should be able to take place in parallel with `FS_WRITEBLOCK` of a different file, and `FS_CREATE` in one directory should be able to take place in parallel with `FS_CREATE` in a different directory.

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 the `ulimit` command.

6.4 Performance and caching

Your file server 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 will **not** cache information between requests. The only information about disk state that your file server should cache in memory between requests is the list of free disk blocks. E.g., your file server should **not** cache inodes or data blocks between requests.

6.5 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.

6.6 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 or inode) should never point to anything invalid (e.g., invalid inode block or data block). 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.

7. Utility functions and utility programs

Your file server must use the utility functions `fs_encrypt`, `fs_decrypt`, `disk_readblock`, and `disk_writeblock` to encrypt data, decrypt data, and access disk. These functions are described in `fs_server.h` and `fs_crypt.h` (automatically included in `fs_server.h`) and are included in `libfs_server.o`. You must use the standard functions `send`, and `close` to send network messages and close network sockets.

(FYI, `libfs_client.o` also includes `fs_encrypt` and `fs_decrypt`, but the client's use of encryption is taken care of by the functions provided in `libfs_client.o` `fs_session`, `fs_readblock`, `fs_writeblock`, `fs_create`, `fs_delete`).

Use `fs_encrypt` and `fs_decrypt` to encrypt and decrypt a data buffer. The size of the encrypted data differs from the size of the cleartext data. `fs_encrypt` and `fs_decrypt` allocate a buffer for the encrypted/decrypted data and return a pointer to that buffer (along with the size of that buffer) . You are responsible for freeing the buffer allocated by `fs_encrypt` and `fs_decrypt` after you are done using it (e.g., `delete [] ptr`).

The encryption functions in `libfs_server.o` and `libfs_client.o` support two types of encryption: CLEAR and AES. CLEAR encryption encrypts the data with a trivial encryption scheme that leaves the data visible; this makes it easier to understand and debug network messages. AES encryption encrypts the data with the AES (Rijndael) algorithm. Both client and server must use the same type of encryption. To specify the type of encryption, set the FS_CRYPT environment variable to CLEAR or AES. In csh or tcsh, you can do this with one of the following lines:

```
setenv FS_CRYPT CLEAR
setenv FS_CRYPT AES
```

In sh or bash, you can do this with one of the following lines:

```
export FS_CRYPT=CLEAR
export FS_CRYPT=AES
```

Use `disk_readblock` and `disk_writeblock` to read and write a disk block (perform only those disk I/Os that are necessary). 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 create an empty file system in the Linux file `/tmp/fs_tmp.<uniqname>.disk` by running the utility program `createfs`. You can run the utility program `showfs` to show the current file system contents stored in `/tmp/fs_tmp.<uniqname>.disk`. Remember to set the execute permission bit on `createfs` and `showfs` (e.g., run `chmod +x createfs 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.

You may set the variable disk_quiet to false to turn off debugging output for disk_readblock and disk_writeblock, and you may set the variable fs_quiet to false to turn off debugging for send and close.

8. Output

Your file server must produce the output mentioned in Section 6.2. It will also (if fs_quiet and disk_quiet are true) 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 protect each call to cout with the cout_lock mutex. This mutex is declared in fs_server.h and is used when libfs_server.o produces output, so you also need to use it whenever your file server generates output.

9. Sockets and TCP

A significant part of this project is learning how to use Berkeley sockets, which is a common programming interface used in network programming. Unfortunately, the socket interface is rather complicated. 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. 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 10 should be 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 fix this, use the MSG_NOSIGNAL flag when calling send; this will cause send to return an error when sending to a connection that is shut down, so your file server knows to stop sending the response.

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.

10. 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 a lot as 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 test case should start with test and end with .cc or .cpp, e.g., test1.cc.

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. Test cases should exit(0) when run with a correct file server.

When we run your test cases, we will start the file server with an empty file system and will input the following passwords:

```
user1 password1
```

```
user2 password2
user3 password3
user4 password4
```

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 13 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.

11. Project logistics

Write your file server in C++ on Linux. Declare all global variables and functions `static` to prevent naming conflicts with other libraries. Use `g++ 4.8.5` to compile your programs.

You may use any functions included in the standard C++ library, including the STL. You should not use any libraries other than the standard C++ library and pthreads. To compile a file server `fs.cc`, run:

```
g++ fs.cc libfs_server.o -pthread -ldl -std=c++11
```

To compile a client application `app.cc`, run:

```
g++ app.cc libfs_client.o -std=c++11
```

You may add options such as `-g` and `-Wall` for debugging.

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`.

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
```

12. 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 auto-grader 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 auto-grader 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.

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

- 0-11: basic functionality
- 12-19,28: error handling
- 20-25: large, serial (i.e., non-concurrent) test cases
- 28-66: start with pre-existing file systems
- 26,29-66: 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 auto-grader will first run a test case with a correct file server to generate the right answers for this test case. The auto-grader 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. However, only the feedback from the first submission of each day will be provided to you. Later submissions on that day will be graded and cataloged, but the results will not be provided to you.

In addition to this one-per-day policy, you will be given 3 bonus submissions that also provide feedback. These 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 1 feedback per 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 6.2.
- 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).

In addition to the auto-grader's evaluation of your program's correctness, a human grader will evaluate your program on issues such as the clarity and completeness of your documentation, coding style, the efficiency, brevity, and understandability of your code, 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 auto-grader score.

13. 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.

The official time of submission for your project will be the time of your last submission. 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.

15. Files included in this handout

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