CS 214: Systems Programming, Spring 2019
Assignment 3: Where's the file?

Warning: This is a complex assignment that will take time to code and test. Be sure to define modules and give it its due time. *Make sure to read the assignment carefully!*

0. Abstract

Git is a a popular current iteration of a series of version control systems. Most large, complex software projects are coded using version control. Version control can be very helpful when working on a single set of source code that multiple people are contributing to by making sure that everyone is working on the same version of the code. If people are working on code in physically separate locations, it is entirely possible that two different people have edited the same original code in two ways that are incompatible with each other. A versioning control system would not allow two different versions of the same file to exist in its central repository, enforcing that any changes made to a file are seen by everyone before they can submit additional changes to the repository.

1. Introduction

You will write a version control system for this assignment. You first need a primer for the vocabulary:

project:	some collection of code that is being maintained also, the directory under which a collection of code resides
repository:	the union of all the canonical copies of all projects being managed as well as metadata and backups/historical data also, the directory under which all managed project directories are located
.Manifest:	a metadata file listing all paths to all files in a project, their state (version) and a hash of the all the contents of each file
history:	a list of all updates/changes made to a given project since creation (is maintained at the repository, but can be requested by clients)
roll back:	change the current version of a project in the repository to be a previous version
commit/push:	upload changes you made to a project locally to the repository, updating its current version
check out:	download the current version of a project in the repository
update:	download from the repository only the files in a project that are newer than your local copy

A version control system consists of some remote server that holds the repository and manages changes to it, and any number of user clients that can fetch projects from the repository and push changes to projects they have checked out. The clients have a local copy of the project they can make changes to, while the server holds the canonical or definitive version of the project.

The overriding mandate of a version control system is to make sure no changes are made to a project in the repository unless the user making the changes to that project has its current version. This can ease difficulties

when working remotely on a shared code base with other people. Rather than, for instance, emailing around copies of code as it is changed, a versioning system will maintain a single, canonical version. A version control system enforces synchronization by requiring that you have the current version of a project before you can submit a change to it. This means it is not possible for two people to check out the current version of a project, make different changes to the same file, and then both submit those changes. The first one to submit their changes will alter the current version of the repository, so the next person who tries to submit changes will need to update to the current version. On updating, they would see the file they have been editing has been altered and would have to integrate their changes with the current version of the file before submitting.

A version control system can also protect a team from development mishaps. If you accidentally delete a file, you can just update from the repository and get it restored. If you have an odd bug and just want to start fresh, you can delete your whole project directory and check out a fresh copy. The version control system also saves all versions of a project as changes are committed to it. Every new version of a project is saved separately in the repository so it is possible to fetch old versions. Given the scope of this assignment, your version control system will have limited functionality and will only be able to roll back (most version control systems can fork a repository to have different development paths, and can roll forward undoing previous roll backs). This would allow a group to start coding, realize that the current design or direction is flawed and roll back to a previous version of the repository. The next check out done would then restore an older version of the project that the version control system had saved.

The server will then necessarily need to support multiple client connections, perhaps simultaneous ones, be able to automatically scan recursively through a project directory and compare files to determine similarity. The server will also need to keep multiple versions of a project so that it can roll back the current version to a previous and send those old files to a client. The client will need to parse commands from the user, scan through a project directory to build a local manifest and know how to communicate with the server to either commit changes make by the user to the local copy up to the repository, or fetch from the server files that it has that are newer versions of files in the user's local copy of the project.

2. Implementation

You will need to write two programs; a “Where's The File” server and client. The server will maintain a repository of projects and the client will commit and fetch updates to and from the server's repository.

The client and server programs can be invoked in any order. Client processes that cannot find the server should repeatedly try to connect every 3 seconds until killed or exited with a SIGINT (Ctrl+C).

Minimally, your code should produce the following messages:

- Client announces completion of connection to server.
- Server announces acceptance of connection from client.
- Client disconnects (or is disconnected) from the server.
- Server disconnects from a client.
- Client displays error messages.
- Client displays informational messages about the status of an operation
  (another operation is required, aborted and reason)
- Client displays successful command completion messages.

2.0 WTF client:

The WTF client program will taken as command line arguments a WTF command and a number of arguments. Most commands require a project name first, some need only the project name. The first command the client is given should always be a configure command where the hostname or IP address of the machine on which the

server is located as well as the port number where it is listening are command-line arguments. After a configure is done, based on the command it is given, the client may send different things to the WTF server. The client program can take only one command per invocation. The client's job is to maintain a Manifest; a list of all files currently considered to be part of the project and to verify that list with the server when asked. Most commands that result in files being sent or received to or from the server have two halves; a command of preparation to get ready for an operation and then a command of execution to do the operation:

update - get the server's .Manifest and compare all entries in it with the client's .Manifest and see what changes need to be made to the client's files to bring them up to the same version as the server, and write out a .Update file recording all those changes that need to be made.

upgrade - make all the changes listed in the .Update to the client side

commit - get the server's .Manifest and compare all entries in it with the client's .Manifest and find out which files the client has that are newer versions than the ones on the server, or the server does not have, and write out a .Commit recording all the changes that need to be made.

push - make all the changes listed in the .Commit to the server side

All other commands are fairly direct; they create or destroy a project, add or remove a file to or from a project, fetch the current version of the entire project from the server or change the current version of the project, or get metadata about the project.

2.1 WTF server:

The WTF server program firstly need to be multithreaded, as it needs to serve potentially any number of clients at once. It should spawn a new client service thread whenever it gets a new connection request. It should not do any client communication in the same execution context that listens for new connections.

Since there will be multiple threads trying to access the files in the repository at the same time, you should have a mutex per project to control access to it. Be sure to lock the mutex whenever reading or writing information or files from or to a project. You do not want to, for instance, send a .Manifest to a client while you're adding a file to it so that the .Manifest sent would be out of date the moment it is sent. Be careful not to deadlock your server's mutexes.

When being started the server takes a single command line argument, a port number to listen on;

```
./WTFserver 9123
```

The server can be quit with a SIGINT (Crtrl+C) in the foreground of its process. You should however make sure that you catch the exit signal (atexit()) and nicely shut down all threads, close all sockets and file descriptors and free() all memory before allowing the process to terminate.

3. WTF Client Commands

The client process will send commands to the server, and the server will send responses back to the client. The server will send back error, confirmation messages, and/or files for each command. All messages sent to the server should result in a response to the client. The client program will take one command at a time and can only perform one command per execution/invocation.

3.0 ./WTF configure <IP/hostname> <port>

The **configure** command will save the IP address (or hostname) and port of the server for use by later commands. This command will not attempt a connection to the server, but insteads saves the IP and port number so that they are not needed as parameters for all other commands. The IP (or hostname) and port should be written out to a ./.configure file. All commands that need to communicate with the server should first try to get the address information and port from the ./.configure file and must fail if configure wasn’t run before they were

called. All other commands must also fail if a connection to the server cannot be established.
Note: if you can write out to an environment variable that persists between Processes, feel free to do so, but all recent feedback has been that security upgrades to the iLabs seem to have obviated this option.

3.1 ./WTF checkout <project name>

The **checkout** command will fail if the project name doesn't exist on the server, the client can't communicate with the server, if the project name already exists on the client side or if configure was not run on the client side. If it does run it will request the entire project from the server, which will send over the current version of the project .Manifest as well as all the files that are listed in it. The client will be responsible for receiving the project, creating any subdirectories under the project and putting all files in to place as well as saving the .Manifest.

3.2 ./WTF update <project name>

The **update** command will fail if the project name doesn't exist on the server and if the client can not contact the server. The client will request the server's current .Manifest for a given project name. When the client receives it, the client will scan through all the local files in its project, generate a hash for every file in its own .Manifest and compare it with the server's, outputting all differences to STDOUT as follows:

U (for upload) and the full path/file name
- for a file that is in the client's .Manifest that is not in the server's, and the client's and server's .Manifest are the same version, or a file that both the server and client have, but the server's hash of the contents and the client's live hash are different, and the client and server's .Manifest version is the same

M (for modify) and the full path/file name
- for a file that both the server and client have, but the server's .Manifest version and file version are different than the client's, and the client's live hash of the file is the same as in its .Manifest.

A (for added) and the full path/file name
- for a file that is in the server's .Manifest, but not the client's and the server's .Manifest is a different version than the client's

D (for deleted) and the full path/file name
- for a file that is not in the server's .Manifest, but is in the client's and the server's .Manifest is a different version than the client's

(U MAD, bro?)
The client should record the action code (M, A or D), version, path and name, and hashcode for all files that need to be modified, added to or removed from the client side in to a .Update file and report the same to STDOUT. At this stage files that need to be uploaded (code "U"), are ignored. If there are simply no differences in version or hash, the client can write a blank .Update and output 'Up to date' to STDOUT.

The other possibility is if the client finds a file that does not fit any of the above criteria and there is a difference between .Manifest version, file version *and* client live hash. This is a conflict and a conflict is ... bad. A conflict means that someone pushed an update to the server (incremented the file's version and server's .Manifest version and modified the hash), but the client has not seen it, and the user made a change to that file locally (so that the hash stored in the client's .Manifest is different than a live hash of the file). This means that contents of the current version of the file, over on the server, are different than the contents of the user's version of the file, locally. If conflicts are found, the client program should not write a .Update, and should output all the conflicting files' paths/names to STDOUT, list them as conflicts and ask user to resolve them before updating again.

Note that the client does not generate a hash for, or record any information for, files in the project directory that are not in the .Manifest file. It is only concerned with files listed in the .Manifest file. If no files need to be altered, the client should write a blank .Update and inform the user that the local project is up to date.

3.3 ./WTF upgrade <project name>

The **upgrade** command will fail if the project name doesn't exist on the server, if the server can not be contacted or if there is no .Update on the client side. The client will apply the changes listed in the .Update to the client's local copy of the project. It will delete the entry from the client's .Manifest for all files tagged with a "D" and fetch from the server and write or overwrite all files on the client side that are tagged with a "M" or "A", respectively. When it is done processing all updates listed in it, the client should delete the .Update file. Note that the client does not make any changes to files in the the project directory that are not listed in the .Update. If the .Update is empty, the client need only inform the user that the project is up to date and delete the empty .Update file. If no .Update file exists, the client should tell the user to first do an update.

3.4 ./WTF commit <project name>

The **commit** command will fail if the project name doesn't exist on the server, if the server can not be contacted, if the client can not fetch the server's .Manifest file for the project or if the client has a .Update file that isn't empty (no .Update is fine). After fetching the server's .Manifest, the client should should first check to make sure that the .Manifest versions match. If they do not match, the client can stop immediatley and ask the user to update its local project first. If the versions match, the client should run through its own .Manifest and recompute a hashcode for each file listed in it. Every file whose newly-computed hashcode is different than the hashcode saved in client's local .Manifest should have an entry written out to a .Commit with its version number incremented. The commit should be successful if the only differences between the server's .Manifest and the client's are:

0. files that are in the server's .Manifest that are not in the client's
(indicating they should be removed from the repository)

1. files that are in the client's .Manifest that are not in the server's
(indicating they should be added to the repository)

2. files that are in both .Manifests, but also have an entry in the client's newly-written .Commit with a higher version number
(indicating the client has a newer version than the server)

If all the differences are only the above cases, the client should send its .Commit to the server (and the server should save it as an active commit) and report success. If however there are any files in the server's .Manifest that have a different hashcode than the client's whose version number are not lower than the client's, then the commit fails with a message that the client must synch with the repository before committing changes. If the client's commit fails, it should delete its own .Commit.

3.5 ./WTF push <project name>

The **push** command will fail if the project name doesn't exist on the server, if the client can not communicate with the server or if the client has a .Update consisting of any files that were modified since the last upgade (i.e., any "M" codes). The client should send its .Commit and all files listed in it to the server. The server should first lock the repository so no other command can be run on it. While the repository is locked, the server should check to see if it has a stored .Commit for the client and that it is the same as the .Commit the client just sent. If this is the case, the server should expire all other .Commits pending for any other clients, duplicate the project directory, write all the files the client sent to the newly copied directory (or remove files, as needed), update the new project directory's .Manifest by replacing corresponding entries for all files uploaded (and removing entries for all files removed) with the information in the .Commit the client sent, and increasing the project's version. The server should then unlock the repository and send a success message to the client. If there is a failure at any point in this process, the server should delete any new files or directories created, unlock the repository and send a failure message to the client. The client should erase its .Commit on either response from the server.

3.6 ./WTF create <project name>

The **create** command will fail if the project name already exists on the server or the client can not communicate with the server. Otherwise, the server will create a project folder with the given name, initialize a .Manifest for it and send it to the client. The client will set up a local version of the project folder in its current directory and should place the .Manifest the server sent in it.

3.7 ./WTF destroy <project name>

The **destroy** command will fail if the project name doesn't exist on the server or the client can not communicate with it. On receiving a destroy command the server should lock the repository, expire any pending commits, delete all files and subdirectories under the project and send back a success message.

3.8 ./WTF add <project name> <filename>

The **add** command will fail if the project does not exist on the client. The client will add an entry for the the file to its own .Manifest with a new version number and hashcode.
(It is not required, but it may speed things up/make things easier for you if you add a code in the .Manifest to signify that this file was added locally and the server hasn't seen it yet)

3.9 ./WTF remove <project name> <filename>

The **remove** command will fail if the project does not exist on the client. The client will remove the entry for the given file from its own .Manifest.
(It is not required, but it may speed things up/make things easier for you if you add a code in the .Manifest to signify that this file was removed locally and the server hasn't seen it yet)

3.10 ./WTF currentversion <project name>

The **currentversion** command will request from the server the current state of a project from the server. This command does not require that the client has a copy of the project locally. The client should output a list of all files under the project name, along with their version number (i.e., number of updates).

3.11 ./WTF history <project name>

The **history** command will fail if the project doesn't exist on the server or the client can not communicate with it. This command does not require that the client has a copy of the project locally. The server will send over a file containing the history of all operations performed on all successful pushes since the project's creation. The output should be similar to the update output, but with a version number and newline separating each push's log of changes.

3.12 ./WTF rollback <project name> <version>

The **rollback** command will fail if the project name doesn't exist on the server, the client can't communicate with it, or the version number given is invalid. This command does not require that the client has a copy of the project locally. The server will revert its current version of the project back to the version number requested by the client by deleting all more recent versions saved on the server side.

4. Methodology

There are any number of ways to code the operations above. There are however some conventions that are common in such situations that can alleviate pressures or problems you haven't come up against yet. In this section we describe some of the common practices you might want to be aware of.

4.0 Simple network protocols

Once you open a socket (to be discussed in class) and connect to the WTF server, you will get a file descriptor. You can treat that file descriptor like any other file descriptor, you can read() and write() from and to it (but you can't f-command it ... see, there is a method to the madness). This is how you communicate with the server. One

thing you must decide before you start writing your sockets is your protocol. Remember, read() and write() just deal in bytes. If you need to download 3 files from the WTF server, for example, you need to know how long each is first. You can't just start reading bytes and somehow know when the first file stopped, since you are just reading bytes. You need to use the same file descriptor to send both commands and data, so you need a rigid set of rules (a protocol) to allow you to separate commands and metadata from actual data since you can't tell them apart just by inspecting the bytes themselves.

A common practice is to, whenever sending data, first send the length of the data and then the data itself. This way you get the number of bytes, and then just read in that many bytes. Presume you define your protocol the following way:

<text command><delimiter>
- if the command is the text 'sendfile' next could be
<number of files><delimiter>
- then, <number of files> number of the following
<filename length><delimiter><filename><size of file in bytes><delimiter>
- then, the bytes for each file, in sequence

So, if you wanted to send two files, 'thing.txt' and 'stuf.txt' and your delimiter is ':'
sendfile:2:9:thing.txt21:8:stuf.txt9:0f009fflll111100JIAlz0&89*1H9s0

The message above broken out and interpreted based on the rules above with the delimiter ":"

sendfile:	I am getting files
2:	I am getting 2 files
9:	first file's name is the next 9 bytes (chars)
thing.txt	.. this is the first file's name
21:	.. this is the first file's length in bytes
8:	second file's name is the next 8 bytes (chars)
stuf.txt	.. this is the second file's name
9:	.. this is the second file's length in bytes

(I've read in information for all (2) files, now I should be getting the files' bytes)

0f009fflll111100JIAlz0	.. these next 21 bytes make up the first file
&89*1H9s0	.. these next 9 bytes make up the second file

You should design your own protocol after looking carefully over all the WTF commands. Be sure to remember that you will need to use the same file descriptor to send commands, metadata and data, so you will need to tell them all apart. As stated above, often the most direct way to do that is to first note the number of bytes something will be, and then to read that many bytes. The reason there is a delimiter after each length is that you do not know how long (how many chars) a given number will be.

A rule of thumb is to use a delimiter for anything that is read in as fairly short text data and number of bytes for anything that is not text data or text data that could be quite long. You don't want to use a delimiter on data because your delimiter might appear as data, which makes things difficult. You often don't want to use a delimiter on something long because you don't want to do a thousand compares looking for your delimiter if you could just count bytes.

4.2 Network Request/Response

Another common rule of thumb for network protocols is to make sure you always get a response for every message. The WTF client should always expect a response from the WTF server. In other words, make sure that you write the WTF server so that it always sends a response to the WTF client for every message. Even if the

WTF server has no data or information to send in response, a simple 'OK' lets the WTF client know that the WTF server has seen the message. This way, you can tell the difference between a command that failed on the server vs a message that got lost.

4.4 File Manifests

Your WTF client will need to determine if the user changed any files in its local copy of the project so that it can tell if it should upload those files to the server or not on a commit/push and if the client's files have been changed and are inconsistent with the server's files on an update/upgrade.

The rule is that the client is allowed to send changes to the server if, for all files the client changed, there were no changes made to those files on the server yet. The most direct way to do this would be download every file in a project from the WTF server and compare them, byte by byte, with the local copies the client has - which is terrible. Instead of doing that, your WTF client will compute a Manifest, ask the WTF server for its Manifest and compare them to determine which files, if any, have changed.

A Manifest is an index consisting of all files in a project (by full path), their current version number and a digest of that file. A digest is some short way of representing the contents of a file. You do not need the *acutal* contents of the file, you just want some code that can be compared to to see if a given file changed. A popular way to implement a digest is to use some type of hash, like AES or SHA. Feel free to use any hash library on the iLabs that suits your purposes. The hash of the contents of the file will change if you change any single byte in the file and hashes can be *much* shorter than the entire file. The WTF server and client should maintain a .Manifest file for each project. A Manifest can be as direct as a version number at the top, a newline, and an entry per line consisting of:

<file version number><<projectname>/path/filename><hashcode>

4.3 Implementation Strategy

Be sure to map out everything first. There are definitely places where you can save coding effort and time by reusing the functionality of some commands in other places. Most of the commands are quite simple. You could write add and remove in a minute or two. Create and destroy are also quite simple. It can help a lot to write currentversion and history early, since they will let you parse through and read metadata to see how things change as you run other commands. For update/upgrade and commit/push it can help to first write out a flowchart of the interactions, which can help to you figure out how to code that functionality and what information and data you need to pass back and forth and what messages you will need in your networking protocol to get it.

As for networking, first get a single-threaded server that you can send a message to and read a response. Then send a message that is a file name and get the server to open that file and send it to you, and have your client side can write it out. Then you can verify if it works or not. Then put your file reading and sending code in to a function, make the function a void */void * and make it a thread. Implement some simple error handling (file does not exist, no read permissions, etc), then modify your client to open a file with a list of file names that it requests from your server and have your server send them all back. Once you have that add functionality so that the client sends a list of files to the server, then the client sends those files to the server and the server saves them on its side. You then have pretty much all the networking functionality you need for the project. Note, you can dovetail this part and the add/remove with a simplified commit/push. Add adds a filename to a list, remove removes it and commit/push sends the file that holds the list of filesnames and then push sends the actual files.

5. Results

Turn in a tarred gzipped file named Asst3.tgz containing a directory named Asst3 with the following files in it:

- A readme.pdf file documenting your design paying particular attention to the thread synchronization requirements of your application.
- A file called testcases.txt that contains a thorough set of test cases for your code, including inputs and expected outputs.
- All source code including both implementation (.c) and interface(.h) files.
  (and no executables or object files)
- A test plan documented in testplan.txt and code to run the test cases you specify
- A Makefile for producing your object and executable files with at least the following targets:
  - all - should build a client executable named "WTF" and server executable named "WTFserver"
  - clean - should remove all executables and object files buildable by the Makefile
  - test - should build all files needed to run your test cases, and an executable named "WTFtest" to run them.

Your grade will be based on:

- Correctness (how well your code works, including avoidance of deadlocks).
- Efficiency (avoidance of recomputation of the same results).
- Good design (how well written your design document and code are, including modularity and comments).

(you must have a group for this project. If you do not have a group entered you will receive a -50 point deduction unless expressly exempted by the Professor. Renew your exemption if you had one previously)

6. Future Work

There are a few common optimizations you can apply for extra credit:
Compression:

Compress old versions of the project at the repository:	+10pts
Compress all files to be sent over the network in to one file:	+20pts
Do the above using zlib or tar library calls (rather than system())	+10pts