README:

AUTHORS
Krishna Prajapati
Rakshay Kanthadai

NAME
server.c - c server file
WTFServer - compiled server executable
client.c - c client file
WTF - compiled client executable
test.c - c test file
WTFtest - compiled test executable

DESCRIPTION

WTFserver:

WTFserver creates a multithreaded server with which the WTF client can interact with using its member functions. The server creates a new thread for every client that connects with it and the thread then executes the command requested by the client. The running thread IDs are stored in a linked list, to which threads are added and removed as they are created and finished with execution. The server creates, holds and destroys projects in its working directory and creates a mutex for each project created that is locked and unlocked as necessary. All the mutexes are stored in a linked list that is created on servers startup from the directories with .Manifests in the current working directory and this list is updated as projects are created and destroyed by client calls (this linked list of mutexes also has a mutex associated with it). While clients are connected, the server cannot terminate on an interruption signal until all clients terminate. On termination, the server goes through the list of running threads and joins them all and closes the sockets from which they originated. Under normal conditions, in which an interrupt signal has not been caught yet, threads will delete themselves from the thread list and free their resources upon completion of their function. There is a mutex for this list of threads as well to allow for synchronization.

WTF

WTF creates one instance of a client which can try to connect to the server given server connection information (which is given through client commands- refer to "USE"). When standing alone, the client can run three of its functions (specified in "USE"), the other ten of which require a server connection. The client interacts with the server by sending data over the socket connection and receiving information back allowing it to access the server's project data, send new project data, and receive new project data.

The client can only send the server new files if it has the most recent version of the project from the server. The client can only receive files from the server if it does not have any changes made to the files locally. These comparisons are all done by hashing the files and comparing their 32 byte hashcodes (using the MD5 hashing algorithm). This is all to ensure that no client can submit new changes to the server's copy of a project without seeing all other clients' changes to the server's copy first- the primary principle of version control.

WTFtest

WTFtest runs through a basic usage of the client-server interface. It forks a process and executes the server on the child process and executes a series of client commands on the parent process similar to what might be done in an actual use of the program. It configures the client, creates projects, destroys projects, adds files, removes files, commits changes, pushes commits, rolls back to a prior version, updates, upgrades, performs history, and performs checkout. Then, the parent process sends SIGINT to the child and waits for it to end to avoid orphaning the process and allows for termination of the server process. The outputs to stdout on both client and server processes are piped to text files. After the test cases finish running and the server is terminated, WTFtest will compare the resulting directories and piped files to our expected results which are generated at the beginning of the executable. All discrepancies will be piped to another text file called diffResults.txt. If this file is empty, the executable will print to stdout that no differences were found between expected and generated results. Otherwise, it will direct the user to refer to the diffResults.txt file for more information about any discrepancies.

MakeFile:

The make commands are clean, all and test.
make clean- removes all .o files and executables
make all - creates all .o files and executables (./WTF and ./WTFserver)
make test - creates a test executable ./WTFtest that tests some basic test cases and compares their outputs to manually generated correct outputs.
make cleantest - removes all directories and files made by WTFtest including the test executable itself.

USE

WTFserver:

./WTFserver <port #>
Starts the server and tries to bind a socket to the port number given as the argument. It may fail due to the port already being in use or for other unknown reasons, in which case it is advised to retry the same port or run on a different port number. Once it starts, the server will be looping infinitely, listening to any and all clients trying to connect to its port and creating a new thread for every successful connection. The thread created for a client will execute that client's command.

WTF:

WTF has 12 total commands. We first describe the ones where a server connection is not required and have put a * at the beginning of them.

* ./WTF configure <IP/hostname> <port>
  Configure saves this IP address or hostname and the port in a .configure file in the client's current working directory. Future function calls that need to connect to the server use these values from the .configure file in order to set up the socket. The IP/hostname is that of the server you wish to connect to and the port is the argument of the running server (refer to server.c use above).
* ./WTF add <projectName> <fileName>
  Add appends a file in the correct form into the .Manifest of projectName in order to start tracking it and appends an A after the version number to indicate that it is tracked locally. Will not work if the fileName does not already exist on the client side. If the file has been removed (has an R as described below in ./WTF remove) and then re-added prior to any commits/pushes, then it is added without an A. If the file is already being tracked in the .Manifest it will not be added again. When added to a .Manifest, a file's version number (0A, unless a locally removed file was locally

re-added), its file path, and its hashcode are written in succession on one new line delimited by spaces.

* ./WTF remove <projectName> <fileName>

Remove removes fileName from the .Manifest of projectName in order to stop tracking it and appends an R after the file's version number to indicate that it has stopped being tracked locally. If the file has been added locally prior (has an A as described above in ./WTF add), the entry is completely removed from the local .Manifest. If the file already has been removed, then nothing happens.

./WTF checkout <projectName>

Receives the most recent version of the project from the server, and writes all files and directories locally to a directory called projectName including the .Manifest. Files sent over are compressed with tar prior to sending by the server and decompressed upon arriving at the client. Checkout will not work if the projectName does not exist on the server, if the client cannot communicate with the server, if client already has a directory named projectName, or if a .configure file does not already exist (implying that ./WTF configure was not run). Checkout locks the mutex of the project while operating and then unlocks after it finishes sending files.

./WTF update <projectName>

Update compares the client .Manifest to the server's .Manifest for any differences. Differences are noted in a newly created .Update file and printed for the user to see. The differences may be the removal of the file, the modification of a file, or the addition of a new file. If there are any changes to the file to be updated (noticed by comparing live file hashes to stored ones in the local manifest), then the updates are written to a .Conflict file instead. The .Update file will be deleted if something is written to the .Conflict file, and if there are no conflicts, then the .Conflict file is deleted. If the client is up to date, meaning there are no differences in the .Manifest versions, then the .Update file is left empty and no .Conflict file is made. Update fails if there is no connection to the server or if the project does not exist on the server or client. Update locks the mutex of the project while it operates and unlocks it upon success or failure.

The following are possible entries to a .Update File:

"A <filePath> <serverHash>"

"M <filePath> <liveHash>"

"D <filePath> <liveHash>"

The following are possible entries for a .Commit file:

"C <filePath> <liveHash>"

"C <filePath> <32 0's for hash>"

./WTF upgrade <projectName>

Upgrade executes the changes noted in the .Update file. You must have ran an update command successfully prior to this in order to have generated a

.Update file and there should be no .Conflict files, otherwise upgrade will not execute. A blank .Update file will simply result in upgrade printing that no updates are necessary. For all the files scheduled to be deleted, upgrade will delete them from the client's manifest, which is done by fetching the server-side .Manifest. For the files to be added and modified, upgrade will send a compressed tar file of the changed/new files and remove those files locally. Then, it will uncompress the tar file locally to add and modify the necessary files. Upgrade will fail if the server cannot be connected to or if the project does not exist on the server or client. The .Update file is deleted upon successful completion of the command. Upgrade locks the mutex of the project while it operates and unlocks it upon success or failure.

./WTF commit <projectName>

Commit compares the client and server .Manifest versions for discrepancies- commit fails and asks the user to update and upgrade if they mismatch. A live hash of the client files on the client's .Manifest is created, and any files with discrepancies between their stored hash in the .Manifest and their live hash are written to a .Commit file. If the file's version also has an A or R at the end (see WTF add and WTF remove), then the file is also added to .Commit. The differences allowed are deletions of files, additions of new files, and modifications to files that already existed in the .Manifest. If these are the only changes found, then the .Commit file is created locally, and it is also sent to the server. The server will rename the .Commit file by appending the client's IP to the file name. Commit fails if the server cannot be reached, if the projectName does not exist, or if there is a non-empty .Update or a .Conflict file on the client side. Commit will also fail if it cannot hash a file that is being modified. Also, if any files on the server's .Manifest are different from the client's version in the .Manifest and have a higher version number, then the commit fails and requires an update/upgrade. Commit locks the mutex of the project while it operates and then unlocks it upon success or failure.

./WTF push <projectName>

Push requires a commit be done successfully prior so that there is a .Commit file on both the client and server side. Then the client sends its own .Commit to the server, the server locks the project's mutex, and the server compares this .Commit to the commit file with the client's IP appended to it. If this comparison yields no differences, then this is the latest commit and push deletes all other pending commits; otherwise the push fails. Push then duplicates the project directory, compresses one copy of the directory as an older version (tar file) and then receives all files that need to be deleted and deletes them in the other copy of the directory (this directory is considered the newest version of the project). Then the server receives all files that need to be added as a compressed tar file and decompresses them in the project directory, deleting old versions of the files in the decompression process. In the process, push prints the files that are compressed on the client side and

the files that are decompressed on the server side. Then the server's project's .Manifest is updated and the client's is made to match. The server then unlocks the project's mutex and sends a success message to the client. Push fails if the server cannot be reached or if the project name does not exist on the server and deletes the .Commit files on both ends on failure. Push also appends the .Commit file to the .History file.

./WTF create <projectName>

For thread synchronization purposes, Create first locks the mutex for the list of project mutexes. It then creates a new node for the project mutex list, since there is a new project that needs to be locked. Create then locks the created mutex for this new project. After locking the two mutexes, Create creates a new project directory on the server with the given projectName and sets up a new, empty .Manifest for it (aside from its version number, 0). The client creates this same directory and receives the empty .Manifest to put in the directory. After a succession creation, Create will unlock the newly created project's mutex before unlocking the mutex for the list of project mutexes. Create fails if the server cannot be reached or if the project already exists on the server.

./WTF destroy <projectName>

Destroy removes the specified projectName on the server. It first locks the mutex associated with the list of project mutexes. Then it renames the project to "\0" so no other thread can find the mutex in the linked list, and then locks the mutex associated with the project so that Destroy waits for any current threads to finish their changes to the project. Next, Destroy expires any commits to the project, deletes all files and subdirectories, including the projectName directory itself, and then returns a success message to the client. After that, Destroy unlocks the project mutex, removes the node associated with projectName from our linked list, destroys its mutex, and then finally unlocks the mutex associated with the list of project mutexes. Destroy fails if the projectName does not exist on the server or if the server cannot be reached.

./WTF currentversion <projectName>

CurrentVersion outputs the manifest version of projectName along with a list of all files under projectName and their file version numbers, as specified in the .Manifest. The client does not require a local copy of the project to perform this command. The command fails if the project does not exist on the server or if the server cannot be reached. The mutex for the project is also locked before the server sends the .Manifest info and is unlocked upon completion.

./WTF history <projectName>

History requests the .History file from the server which contains all the information of previous pushes (essentially a list of pushed .Commit files and the version number they were committed on). History fails if the server cannot be reached, or if the project does not exist on the server. History does not require the client to have a local copy of the project. The server also locks the mutex of the project after it receives the command and finds the project mutex. The mutex is then unlocked after a successful send of the .History file.

./WTF rollback <projectName> <versionNumber>

Rollback sets a project given by projectName to its previous version specified by versionNumber. All previous versions are stored as tar files on the server so the server deletes the project directory and all versions other than the version specified by versionNumber. Then it decompresses the tar file into a projectName directory for use. Rollback locks the mutex of the project while it operates and then unlocks it after successful rollback. Rollback fails if the project does not exist on the server or if versionNumber is not less than the project's manifest version.

WTFtest:

./WTFtest

WTFtest should be used as a basic test to see if the client can connect and run all the commands necessary. The executable runs all 13 client commands as well as the server as described above in DESCRIPTION. Any failures of any commands will be written to diffResults.txt and a print statement will notify the user of discrepancies between expected and generated results. Otherwise on success a blank diffResults.txt is generated along with all the project directories and piped stdout files. The user will also be notified that there are no discrepancies. The executable should result in two project directories that are in the client folder and one project directory that is the server's copy in the assignment root folder. The two projects in the client directory are testProject with a manifest and two files (one of which has some text) and destroyThis, with a new manifest. The project directory in the assignment root folder should be testProject containing a .History, a .Manifest, a tar.gz file and two text files. The executable also generates hidden result files and directories to compare the testcase outputs. These results files have the expected results hard-coded in. All generated files from this command can be removed by using the command ">make cleantest"

DESIGN

Here we describe some design choices for different functions and commands in depth.

.Manifest Files:

Manifest files are critical for describing a repository's data succinctly. Our manifest files hold a version number as their first entry, then a new-line separated list of files with the following format: <versionNumber> <filePath> <hashCode>. Manifests can be edited to have an A or an R after the version number on local client manifest copies in order to differentiate local adds and removes from the server adds and removes. Manifests can also be converted to AVL trees (refer to AVL Trees section below).

Other Files (.Update, .Conflict, .Commit, .History):

Other files are made throughout the program but all bear a resemblance to the .Manifest format. .Update, .Conflict, and .Commit all follow this general form: <Letter> <filepath> <hashCode>. This is just the .Manifest format but with version number replaced with a letter signifying what is happening with the file associated with this line (Delete, Modify, Add, Conflict). Except for .Conflict, they all start with a version number and then a newline separated list of these entries. .Conflict only has these newline separated entries. Their similarities allow these files to also be converted to AVL Trees (refer to AVL Trees below). The .History is slightly different as it is just a series of .Commit files separated by newlines and the version number at which the commit was executed.

Mutexes:

We needed a dynamic way to create and keep track of a mutex for every project so we made a linked list with project name and a mutex stored in the nodes. This linked list is created from the directories in the WTFserver working directory upon startup and maintained through creates and destroys, which add and remove from the linked list. We use this list to check if projects exist on the server for all the client.c functions that require that check. If a project name is not in the linked list, we always terminate and notify the user that the project does not exist on the server. We also have a global mutex associated with the linked list itself that we lock whenever editing or searching the linked list, and then unlock when the search or edit is complete. A lock is needed to make sure that nodes always are connected properly when searching and editing. There is one more global mutex, which is the mutex for the list of threads. This mutex is locked whenever a thread is added to the list, or when a thread removes itself from this list, and it is unlocked when the add or remove operation is complete. This is done to avoid the issue of multiple threads terminating and potentially ruining linked list connections due to the freeing of dependent nodes.

Threads:
We needed a way to allow multiple clients to access the server and run their commands at the same time so we needed to let every new client connection be a thread that runs independently. The running thread IDs then needed to be stored in a linked list (dynamically sized) so that upon interrupt, all running threads could be neatly joined and the sockets they are associated with neatly closed. Upon terminating normally, the thread deletes itself from the linked list of currently running threads and then detaches itself. This linked list of threads also has a mutex associated with it so as to allow synchronization from multiple finishing threads.

AVL Trees:
Sometimes, we want to manipulate the .Manifest file or get information out of it. Thus, we have a function that takes a string form of the .Manifest file and creates an AVL tree out of it in which each node is an entry in the .Manifest. This allows us to quickly search the .Manifest by file name for the commit and update functions. In addition, for push, this data structure allows us to traverse a .Commit file as an AVL tree (due to its similarity to a .Manifest) and apply its changes to a .Manifest, and then convert this AVL to an updated .Manifest file.

Hashing:
Our program hashes using the MD5 protocol of OpenSSL to generate a 32-byte hashed string for a given file.