Where's The File? (CS214 // Assignment 4)

Contributors: Seth Karten & Yash Shah

Overview

Usage

Call all commands from the directory with the Makefile. For example to run the server and client, run:

```
make
bin/WTFserver <PORT>
bin/WTF <args>
```

To run the test cases, run:

```
make test
bin/WTFtest
```

To remove all built files, run:

```
make clean
```

Thread Synchronization Requirements

During the execution of our multithreaded server, we do not want to result in race conditions. To avoid this, we use a table (linked list) of project mutexes. In order for a thread to access any of the contents of a project, it first must fetch the lock for that particular mutex from the table.

First, let us briefly mention a few remarks about the table. The table is a linked list of structs. Each struct contains a mutex, the project name, the number of commit files stored for this project in the server, and a pointer to the next struct in the list. The table itself is locked by a mutex.

So in order to fetch the lock for any project mutex, the table mutex must be unlocked. Thus, the function that fetches the lock for any project mutex first checks that the table is open, then locks it to prevent any other thread from editing the table while it looks up a project mutex. The function unlocks the table before returning a pointer to the project mutex.

We must now make a remark about the individual project mutexes. We want to know how many project's mutexes are locked at any given time so we create a number accessed counter that represents this. When a

thread locks a project's mutex, it first locks the access counter's mutex, then it increments the counter, then unlocks the access counter's mutex. Similarly, when a thread unlocks a project's, it locks, decrements, and then unlocks the access counter mutex, ensuring that an accurate number of project mutexes are counted at one time.

The aforementioned access counter is important when the table of project mutexes needs to be edited (e.g. when a project is destroyed or created), we implement a trickle system. First the thread that is going to edit the table locks the table mutex. Then the thread waits until all the threads trickle out, that is, the thread waits until the access counter's value is 0. When it is 0, we know that there are no threads accessing any projects. Thus, we can edit the table safely. When the query is resolved, the table is unlocked and other threads may begin to access the projects from the table again.

When the server receives communication from the client, it creates a new thread to deal with the client so that the server is always listening. The threads for the server are managed by a linked list with a cap of QUEUE_SIZE. When there are 1+QUEUE_SIZE threads in the linked list, the oldest thread is joined from the list so that there never exceeds 1+QUEUE_SIZE threads on the server. We chose QUEUE_SIZE=20 for reasonable client handling but our server can handle many more threads if the parameter is changed.

Server Signal Handling

The server can be quit with a SIGINT (Ctrl+C) in the foreground of its process. The server catches this signal with sigaction and calls a termination handler function. The termination handler joins all threads from the linked list of threads, thereby removing all the threads. Afterwards the linked list data structure is freed and all files created by the server are deleted.

Compression using zlib (+40)

We use the zlib library to do the following compression optimizations:

1. Compress old versions of the project at the repository
2. Compress all files to be sent over the network in to one file
3. Do the above using zlib (rather than system())

To compress directories using zlib, we use the following protocol. Each file is recursively added to a dynamically sized buffer to contain the file/directory name, flag that it is a directory or a file, file size, and file contents. (Obviously if it is a directory that we are storing we do not include file size and contents.) Afterwards, the buffer is compressed using the zlib deflate function. The file is then processed according to what we need it for (either network communication with the protocol in the next section or written to a file for backup). Decompression is also done with zlib using inflate on the compressed buffer. Afterwards, the decompressed buffer is parsed for the file/directory name, flag that it is a directory or a file, file size, and file contents for a specific application depending on what function needs it. For example, checkout decompresses the file and recreates the directory structure.

Network Communication Protocol

We establish three distinct types of messages between the server and client. Short messages, long messages, and file transfer.

Short messages are three character long strings that are sent to transmit a command from the client to the server. The server then returns "Ok!" or "Err" if the message is successful or not, respectively.

Long messages are 30 character long strings that transmit long success or error messages. These are sent typically at the conclusion of communication from the server to the client. The client prints these to the terminal when received.

File transfer messages are sent in the following manner:

```
<length size of decompressed file><size of decompressed file><length of size of
compressed file><size of compressed file><compressed file>
```

The length of the size of the decompressed file and the length of the size of the compressed file is a three digit string representing a number between 0 and 999 so we can send files up to size 10^1000 which far exceeds the actual size of a file for our application. Thus, we can send files of arbitrary size with our communication protocol.

Server and Client Messages to the Terminal

Our client and server each respectively print messages to their terminal

- 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.
- Client displays successful subcommand (Important helper functions) completion message.
- Server shutsdown
- Server terminates thread

Efficiency Notes

We read the file manifest into a struct to only read a manifest once during our implementation. We also only send over modified files in a compressed buffer during push.

General Implementation Notes

Our client commands are implemented as detailed in the project guide. (Note: We start our manifest at version=2 for our configuration checking.)

We detail the structure and modularity of our .c files below:

- client_cmds: Contains functions to execute the client commands as well as setup communication with the server.
- client_main: Manages parsing command line and error handling for client. Has main function for WTF executable.
- commit_utils: Contain helper functions to use with commit command.
- compression: Contains functions for compressing/decompressing to/from a single file using zlib.
- fileIO: Contains functions for general file inputs/output as well as sending and receiving files over a socket.
- flags: Stores debugging flags, such as a TRUE and FALSE enum.
- manifest_utils: Contains functions for reading a manifest into our defined struct as well as comparing two manifest structs.
- server_cmds: Contains functions to execute commands sent by client.
- server_main: Manages startup and termination of the server. Manages client-server communication. Has main function for WTFserver.
- threads_and_locks: Contains functions for managing mutexes for projects and the project table.
- WTFtest: Runs test cases.