CS352 Spring 2011 Project 3 – DUE DATE: Apr. 28th, 2011 at 11:55 PM

MULTI-THREADED DROPBOX SERVER

PROJECT DESCRIPTION

For the final phase of our projects, you will be implementing a Dropbox-like client/server application that can synchronize files between multiple clients. To do so, you will need to do the following:

1) Modify your client from Part 2 to be able to accept **`chunk`** messages from the server and write/delete files on disk.
2) Add a few new types of messages for synchronization and error-handling purposes.
3) Modify the client and server applications to be **multi-threaded**. To do this under Linux, you must familiarize yourself with POSIX threads. On most systems, see `/usr/include/pthread.h`
    1. Design your server application such that newly accepted connections can always be serviced by a new thread(s).
    2. Modify your client application to maintain two open connections to the server. Your server may utilize two server sockets if you wish:
        1. One connection for sending updated files/metadata to the server.
        2. One connection for receiving updated files/metadata from the server.

****** Please note that the behavior specifications for the second project still hold unless otherwise noted in this document. Read this document in its entirety before beginning your implementation ******

SYNCHRONIZATION

In a multi-threaded program, you must worry about synchronization of data structures between threads. Your server application must maintain a roster of known files and their current status. We will adopt a very simplistic policy for resolving collisions between clients on the same file:

1) The first client to connect to update a file locks it for writing (see `pthread_mutex`)
2) Clients that connect to update a file while it is locked receive an access denied message
3) When a file is finished updating, the server shall send the updated file to all other clients, overwriting any local copy they have (even if it has been updated in the interim!)

PROTOCOL

Message types will be similar to the previous two projects, with a few modifications.

Message Types

0x0001: File Chunk
0x0002: Status Message
0x0003: Finished / Close File
0x0004: Abort File
0x0005: Delete File
0x0006: Request File Update

File Chunk messages will be preceded by a variable-length header with the following fields:

- message type (2 bytes, little-endian)
- md5 digest (16 bytes)
- chunk size (4 bytes, little-endian)
- filename length (2 bytes, little-endian)
- filename (`filename length` bytes)

Finished/Close File, Abort File, Delete File, and Request File Update messages shall consist of the following fields:

- message type (2 bytes, little-endian)
- md5 digest (16 bytes)
- filename length (2 bytes, little-endian)
- filename (`filename length` bytes)

Status Message messages shall consist of the following fields:

- message type (2 bytes, little-endian)
- message code (2 bytes, little-endian)

Message Codes

0x0001 – Status OK

0x0002 – Resend / Checksum Incorrect

0x0004 – Resend / Invalid Chunk Size

0x0008 – Abort after 3 incorrect checksums

0x0010 – Abort after 3 invalid chunks

0x0020 – No file to delete

0x0040 – Abort – File Locked

0x0080 – Abort – File In Use

~~0x0100 – Request File Update~~ **COVERED BY MESSAGE TYPES ABOVE**

0x8000 – Keep-alive message

COMMAND LINE FORMAT

Client Program:

./client <remote host> <send port> <receive port> <folder name> <chunk size>

Server Program:

./server <send port> <receive port>

PROGRAM BEHAVIOR

Client:

Upon connecting to the server, the file shall expect the server to send a META-DATA file containing the state of the server's own META-DATA file. The client will compare the server's META-DATA file to its own, taking the following actions as necessary:

1) If the server has a newer version of the file, the client shall send a `Request File Update` message to the server for each such file
2) If the client has a newer version of the file, the client shall attempt to initiate a transfer of the updated file to the server for each such file
3) If the server has a file that the client does not have in its META-DATA, then the client shall send a `Request File Update` message to the server for each such file. The client shall process updates by writing these files to disk. If the client receives an `Abort – File In Use` status message, the client will abort its attempt to receive an update from the server.

4) If the client has a file in its META-DATA that the server does not have, the client shall attempt to initiate a transfer of the updated file to the server for each such file.

The client shall monitor the local disk for changes in files (updated metadata). Upon detecting a change, the client shall update its local META-DATA file with the information of the modified file. The client shall then attempt to transmit the changed file to the server (either by transmitting chunks or with a `Delete File` message). If the server replies with an `Abort - File Locked` or a `No File To Delete` message, the client shall abort the transfer and give up on trying to send an update. The client will update its own META-DATA file to reflect the changes on its own disk anyway.

The client shall listen for messages from the server, and shall handle `File Chunk, Status Message, Finished / Close File, Abort File,` and `Delete File` messages. Upon receiving a `File Chunk` message, if the client cannot open the file for writing for some reason, it shall reply to the server with an `Abort - File In Use` message. Upon successful receipt of a file from the server, the client shall update its META-DATA to reflect the time at which it finished receiving the file. The client shall NOT attempt to issue an update of the file to the server upon closing it. Upon receipt of an `Abort File` message, the client shall close the file and delete it from disk. Upon receipt of a `Delete File` message, the client shall delete the corresponding file from disk, or if it does not exist, shall respond with a `No File To Delete` status message.

The META-DATA file format for the client shall remain the same as in the second part of the project.

Server:

The server must be able to accept connections from an arbitrary number of clients, and will accept two connections from each client. It is **HIGHLY** suggested that two threads (one for *sending* updates, one for *receiving* updates) be created per client. This will help to avoid deadlock issues.

In addition to the META-DATA file format that was specified in the second project, the server shall maintain the following information for each file:

- A lock for that file (see `pthread_mutex`)

The server shall maintain a list and count of all connected clients in a synchronized data structure. In this list, the server shall maintain a data structure for each connected client:

- Client address

- A queue of updated files to be sent to clients

When a client connects or disconnects, the master list shall be updated accordingly by adding or deleting an entry.

The server shall maintain a synchronized META-DATA data structure in memory, which will be augmented with a locking mechanism (see above). Upon receiving a chunk file from a client, the server thread corresponding to that client shall check to see if the file is in its META-DATA. If so, the server will attempt to obtain the lock on that file. If successful, the server thread shall open the file for writing and process the file as in the previous project. If unsuccessful, the server shall reply to the client with an `Abort - File Is Locked` message. If the file does not exist in the server's META-DATA structure, the server shall create an entry and process updating the file as usual. If the server receives a Delete File message from a client, and finds that the file is in use by another client (locked), then the server shall reply to the client with an `Abort - File Is Locked` message. If the server receives a Delete File message from a client, and the file does not exist in the server's META-DATA structure, the server shall reply with a `No file to delete` message.

All operations on the server's META-DATA and client list data structures should be performed in a thread-safe manner (i.e. these structures should have locks associated with them). See *pthread_mutex_lock*, *pthread_mutex_trylock*, and *pthread_mutex_unlock* for details on blocking and non-blocking locking methods.

When a server *receiving* thread for a client finishes updating a file, it shall attempt to send the updated file to all other connected clients: The *receiving* thread shall wait on the lock for each other connected client's data structure, and upon obtaining it add an entry into the queue of pending updates, unlocking immediately upon insertion. If a file is already in the update queue, it shall not be added again. A client's *sending* thread will check this queue by locking it, dequeuing one item, then unlocking it whenever it is not in the process of sending data to a client. The *sending* thread shall NOT maintain its lock throughout the file transfer process. If a *sending* thread receives an `Abort - File In Use` message from a client, it shall give up on issuing the update to that client.

MULTITHREADING

For a tutorial on POSIX threads, see:
http://randu.org/tutorials/threads/

SUBMISSION

- Submit your project in .tar, .gz, .7z, .rar, or .zip format.
- The submission should contain:
  - Makefile
  - README / writeup
  - Source code & related files
- The submission should **NOT** contain:
  - Object files
  - Large test files
  - Files related to version control

GRADING

60 pts: Proper Functionality

- Builds successfully
- Client handles updates properly
- Client issues updates properly
- Server handles updates properly
- Server manages synchronization correctly
- All error conditions handled properly

20 pts: Code Quality

- Proper memory handling (use Valgrind)
- Proper implementation of protocol (will be checked with Wireshark)
- Proper use of sockets (always shutdown correctly)

10 pts: README / writeup

- Explain how to build and run your project
- Explain design decisions you made (code structure)
- Discuss difficulties you encountered

10 pts: Hand-in Organization

- Proper conformance with submission requirements above
- Makefile includes clean & make all targets
- Code is organized in an easy to read way (division of functionality across appropriately different functions & source files).

EXTRA CREDIT

10 pts: *Worker thread pool* – Many web servers use a *pool* of worker threads to handle incoming requests. Implement a thread pool to handle incoming requests, with a work queue to keep track of pending requests and to issue them to workers in FIFO order.

20 pts: *Multithreaded client & Server* – Allow the client to send multiple files to the server at once. Allow for parallel uploads of files at once rather than sequential uploads. You may implement multiple connections to the server to do this (adds server complexity), or interleave chunks of updated files within a single client connection. In either case, the server will need to be able to process chunks from clients that come from different files at once. This may add significant complexity to the implementations of both your server and your client.