DESIGN DOCUMENT

Author: Niland Sshumacher
Student ID: 1612833
email:naschuma@ucsc.edu

PROJECT DESCRIPTION

This multi-threaded linux http server implements a GET, PUT, and HEAD request functionality via http/1.1. The server can send and receive files of any type as well as http response codes 200, 201, 400, 403, 404, 500, and 501. The server works with both curl and ncat, and could possibly work with other programs using http/1.1.

PROGRAM LOGIC

The program must be passed a mandatory port field, and two optional logging and num-thread fields(specified by -l and -N respectively). The server first creates the number of threads specified with a default of 4. Then a listening socket is created using the port given as the argument and enters a loop to pass accepted connection sockets to worker threads. Once accepted by a worker thread, the server creates a ‘persistent connection’ via a loop in which the socket waits to receive http requests. The server only exits the connection if it receives a ‘400 bad request’. If the logging option was specified the server will write its responses to each client in a logging file with the name specified by the user.

For a HEAD request the server returns the file content length. For a GET request the server returns the file content length following with the file. For the PUT request the server recvs bytes equal to the content-length provided by the header.

The server can handle multiple requests using the same socket as long as the server is never forced to send a 400 bad request, and the receiver never closes the connection on their end..

DATA STRUCTURES

Httpserver uses two 4mb buffers to send and receive http headers and to read and write files. The program can also access entire files(including the log file) that are in the same directory, via the open() and write() system calls. The program also declares several character arrays to store the premade responses.

FUNCTIONS

- void handle_connection(int connfd)
  Enters an infinite loop which waits to receive data from the accepted socket. Upon receiving data it attempts to parse it as a http header and then call a reply function according to the header type. It calls reply_get() for a get header, reply_put() for a put header, etc. A function must return 400 for the function to break the loop and return to wait for another client to connect.

- Int get_content_length(char *buffer)
  A small 8 line function that finds the content-length field in a received header, and returns the content length. If no content length, it returns -1.

- int invalidFilename(char *filename)
  Small function to simply check the filename only contains alphanumerics and is of length 15. return 0 if so, 1 if not.

- int reply_put(int connfd, int content_len, char *filename, char *buffer);
  This function reads a 4mb into a newly opened file identified by the header until it has read all bytes sent or it has read bytes equal to the content length specified by the buffer. It then returns with 201. If the file cannot be overwritten, it returns 403. If the number of bytes written does not match up with the content length it returns 400 and closes the clients connection. If it has an error writing the file it returns a 500.

- int reply_get(int connfd, char *filename, char* buffer)
  This function first gets the file size of a file via stat() and then opens that file specified by the header, and then sends a 200 response followed by the file in 4 mb chunks at a time which are put into a buffer before being written to the socket. If the requested file does not exist, the server returns 404. If the requested file does not have read permissions the server responded 403. If the requested file is a directory, the server sends 400. If the server has an error reading the file it does not return 500 as the client would likely just write the 500 response to the file, so instead the function only returns.

- int reply_head(int connfd, char *filename)
  This function acts very similarly to reply_get except that the file does not need read permissions, and the function never sends the file back, it just sends a 200 along with the file's size

- lock_file(char * filename, pthread_mutex_t *p_mutex, pthread_cond_t* p_cond_var)
  Attempts to lock the given filename by adding it to the list of locked filenames, if the file is already locked it waits until it receives signal that a file was unlocked, and then checks again if the file is still locked.

- unlock_file(char * filename, pthread_mutex_t *p_mutex, pthread_cond_t* p_cond_var)
  Attempts to unlock the given filename found in the list of locked filenames. Upon free the filename, it sends a broadcast to all threads waiting on a locked file.

- log_success(char *request_type, char *filename, char *hostname, int content_len)
  First locks the log, then writes the appropriate log entry to the log before unlocking the log.

- Log_error(char *request, int code)

Same as log_success, except it creates a different form of entry.

- Get_request(pthread_mutex_t* p_mutex)
  Used by worker threads to get the connfd of an accepted socketed, so that the thread can handle the connection of that socket, and unlock the mutex. Else, the thread holds the lock until it reaches the wait in the request loop.

- Add_request(int connfd, pthread_mutex_t* p_mutex, pthread_cond_t* p_cond_var)
  The function that the main thread uses to send to a connections socketfd to a list of requests, and signal to a worker thread that a request has been added.

- void* Handle_requests_loop(void * data)
  Where a worker thread waits to be woken when a new connection is made

- parseArgs(int argc, char* argv[])
  Takes in the arguments given to main, and checks that the flags are correct, and that the port is valid, the logfilename is valid, and the number of threads is valid

- readWriteBytes(int x, int read_fd, int write_fd)
  Writes numbytes from the read_fd to the write fd

- get_hostname(char * buffer, char *hostname)
  Simply retrieves the hostname from the http header using strstr()

QUESTIONS

For Assignment 2, please answer the following questions:
●Using `curl` and your original HTTP server from Assignment1, do the following:
○Place eight different large files on the server. This can be done simply by copying the files to the server's directory, and then running the server. The files should be large enough so that a single GET request would take at least thirty seconds to finish. We suggest that you start creating files with100 MiB and increase or decrease the size according to the performance of your machine.
○Start httpserver.

○Start eight separate instances of the client at the same time, each one GETtingeach of the files, and measure (using time(1)) how long it takes to get the files.The best way to do this is to write a simple shell script (command file) that starts eight copies of the client program in the background,by using & at the end (seemore about this in the Hints section).

●Repeat the same experiment after you implement multi-threading.Is there any differencein performance? What is the observed speedup?

I could not notice any speedup probably b/c I wasn't creating big enough files. But there indeed should be speedup if the files are large in size.

●What is likely to be the bottleneck in your system?How do various parts (dispatcher,worker, logging) compare with regards to concurrency?Can you increase concurrency in any of these areas and, if so, how?

The bottleneck of the program would most likely be the reading and writing of large files, which could be sped up by having another set of worker threads to give parallelism to read/writing. Another bottleneck would be that there is only one thread creating accepting sockets, when there could be many.

●For this assignment you are logging only information about the request and response headers. If designing this server for production use, what other information could you log? Why?

You could also log the time of the request, so that you would be able to determine times your server is used.

●We explain in the Hints section that for this Assignment, your implementation does not need to handle the case where one request is writing to the same file that another request is simultaneously accessing. What kinds of changes would you have to make toyour server so that it could handle cases like this?

My program actually does handle the case of requests accessing files simultaneously. It achieves synchronicity by creating a list of locked files which are added and removed using mutexes, and assures a already locked file will not be accessed by another thread.

TESTING

I made 3 different test files each that ran around 500 curl requests in parallel, each I used to test the multi-threading and synchronization of writing and reading files. One tested correct response and logging to a get, head, put, and fail response. Another tested accessing multiple different files using put. And the last one was testing speed up from different numbers of worker threads, however I wasn't able to find any noticeable speedup.