Design Document: Multi-threaded Web Server

Robert Hu

CruzID: ryhu

1 Goals

The goal of this programming assignment is to modify the previously created simple web server so it utilizes threading and logging. The original web server handled HTTP\1.0 requests. All requests needed to handle content length as well as some http response codes and now had to handle multiple client connections. Using synchronization techniques, the web server should be able to process and handle multiple requests in parallel rather than in sequence like the original web server. In terms of logging, the web server should only log when the -l flag is specified in the initial command. If the flag is listed and the log file is valid, then all logging will be sent to this one file. Since all threads will be writing to the same log file, there needs to be appropriate synchronization techniques to manage access to log files.

2 Design

2.1 Original design

- Parsing using strtok to separate received messages and realloc to allocate memory
- Send/recv and write/read to input from and output to data and/or failures to sockets

2.2 Multithreading

- Check for the *-N* flag
    - If present, set the number of threads to the number that follows the flag
    - Otherwise set the number to the default of 4
- There needs to be the same number of worker threads as are specified by the number of threads. Using *pthread_create()* for both creating the n worker threads as well as the single dispatcher thread using
    - *pthread_create(&dispatcher_thread , NULL, dispatcher, (void*) &arguments)*
    - *pthread_create(&worker_thread[i] , NULL, worker, (void*) &arguments)* inside of a for loop from between 0 and N.
- Synchronization needs to be there to modulate and control access between shared resources and access to the entries in the listen queue that are to be released by accept(). Code to use
    - *pthread_mutex_t* to create the initial locks that will be used
    - *pthread_mutex_init* to initialize the values to the locks that are used
    - *pthread_mutex_lock* and *pthread_mutex_unlock*
        - This code will occur around accesses to the log file, if it exists

        - Otherwise, the unlocks and locks will maintain access to the available resources for requests
- New functions *dispatcher()* and *worker()* to handle what each thread will do
    - *Dispatcher()* will listen for new connections and distribute each new request to a different worker thread and lock and unlock appropriately
    - *Worker()* will wait and sleep until the dispatcher sends them a request to do. After processing the request, the worker thread will return to sleep.

2.3 Logging

- Check for the *-l* flag
    - If it exists, then open/create a logging file that is specified in the command
    - Otherwise no logging will be done
- Logging will need to handle failed cases as well by logging a Fail along with the error response code that follows
- In order for Logging to be done, only one thread may access the log at a time. This can be achieved by using a *pthread_mutex_t* to lock and unlock before and after the log file is accessed by a thread. This will prevent other threads from accessing the log file while another file is actively logging.
- Format of logs
    - Success: PUT abcdefghij0123456789abcdefg length 36 00000000 65 68 6c 6c 3b 6f 68 20 6c 65 6f 6c 61 20 61 67 6e 69 20 3b 00000020 65 68 6c 6c 20 6f 54 28 65 68 43 20 72 61 29 73
        - length is Content Length
    - Fail: FAIL: GET abcd HTTP/1.1 --- response 400\n

2.4 Data structure

- struct Entry containing
    - *char *fileName* - name of the file gotten from the parsed request header
    - *long fileSize* - size of the file gotten from fstat and the struct stat
    - *char *requestType* - request type such as GET and PUT
    - *int status* - http status code such as: 200, 201, 400, 403, 404, 500
    - *int socket* - socket of the client that was accepted by the server socket