Assignment 2: Multi-Threaded HTTP Server W/ Logging - Design Document

Goal:
The goal of this assignment is build on top of the httpserver from Assignment 1 to make the httpserver multithreaded as well as feature logging.

Handling arguments:
Two new flags will be introduced to this assignment. The new flags are -N %d and -t %s where -N defines the number of threads that should be created for the pool of threads, and -l specifies the name of the log file.

By default, the number of threads will be 4 unless specified by -N

In order to handle the flag options, getopt() will be used in the main function to parse through argv and get the flags

Additionally optind will be used to trim off the flags so that the only arguments will the server name/address and port number, which will make things a lot easier as flags can be anywhere in the cmd argument.

```
| //First define the default number of threads                          |
| int nThreads = 4;                                                     |
| //Now use getopt() to get the flag arguments                          |
| while((opt = getopt(argc, arg "N:l:")) != -1) {                       |
|    switch(opt) {                                                      |
|       case 'N':                                                       |
|          nThreads = atoi(optarg);                                     |
|          break;                                                       |
|       case 'l':                                                       |
|          logFileFD = open(optarg, O_RDWR| O_TRUNC| O_CREAT, 0666);    |
|          break;                                                       |
|    }                                                                  |
| }                                                                     |
|                                                                       |
| //Now we must define a new argv that has all of the flags taken out   |
| //and the only arguments will be server name/address and port num     |
| std::vector<char*> new_argv;                                          |
| for(;, optind < argc; optind++) {                                     |
|    new_argv.push_back(argv[optind]);                                  |
| }                                                                     |
```

Now that the flags are set up and cleaned up, setting up the threads will be the next priority.

Semaphores and mutexes will later be used for avoiding race

conditions for the threads, so they will be declared as well.

```
//These will be declared as global variables
sem_t empty;
sem_t full;
pthread_mutex_t request_mutex      = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t globalOffset_mutex = PTHREAD_MUTEX_INITIALIZER;

//Now these will go into the main function
//Firstly, the pool of threads will be created
pthread_t pool[nThreads];
//Now the semaphores must be initialized just like in the consumer
//producer algorithm as the algorithm will be used for this
//assignment
sem_init(&empty, 0, nThreads);
sem_init(&full, 0, 0);

//Now the threads will be created
for(int pos = 0; pos < nThreads; pos++) {
   pthread_create(&pool[pos], NULL, fn, (void *) (intptr_t) pos);
}
```

When the threads are created, they will each go to the function fn and wait on the semaphore full. Meaning that they will wait if there are no connections passed from the dispatcher thread.

The dispatcher thread will be the infinite for loop from Assignment 1 that will continue to listen to requests, then that request will be pushed into the vector fd_buffer and signal to the threads that there is a request available.

```
//The vector will be declared globally as it is accessed by all
//threads
std::vector<int> fd_buffer;
//Just like in Assignment 1, this will be in the main function
for(;;) {
   //This line is essentially from Assignment 1
   int fd = accept(connection);
   //Wait if all threads are preoccupied
   sem_wait(&empty);
   //lock the mutex when assigning a connection to the vector
   //to avoid race conditions when a thread takes from the vector
   pthread_mutex_lock(&request_mutex);
   fd_buffer.push_back(fd);
   pthread_mutex_unlock(&request_mutex);
   //Signal to the threads that a connection is available
   sem_post(&full);
```

```
| }                                                                    |
------------------------------------------------------------------------
```

Now that the method for assigning connections to threads is working, the threads must actually accept that connection and begin working

```
------------------------------------------------------------------------
| void *fn(void* arg) {                                                |
|   //Sleep until the dispatcher signals that there is a connection    |
|   //available in the fd_buffer                                       |
|   sem_wait(&full);                                                   |
|   //Lock the mutex when taking a connection from the fd_buffer       |
|   pthread_mutex_lock(&request_mutex);                                |
|   int fd = fd_buffer.back();                                         |
|   fd_buffer.pop_back();                                              |
|   pthread_mutex_unlock(&request_mutex);                              |
|   //Now execute the connection with the thread. Note this function   |
|   //is originated from my Assignment 1 code.                        |
|   parseHeader(fd);                                                   |
|   close(fd);                                                         |
|   //After finishing the request and closing the fd, signal an empty  |
|   //indicating to the dispatcher that there is space in fd_buff      |
------------------------------------------------------------------------
```

Now that multithreading is working, logging must be implemented to the PUT and GET requests so the thread can log while executing the reply_GET and replyPUT methods.

For logging GET/PUT requests and errors, a global offset must be used for determining at which spot a thread is going to write to for the log fd.

Additionally, a mutex must be used to avoid race conditions when the thread is using the global offset to print as having 2 threads using the same global offset to pwrite() would not be ideal. Global offset is also a counter, so it must be incremented once at a time.

Sprintf() and pwrite() will be used. Sprintf will need a buffer of a predefined size, which will have to be calculated.

Start off with logging GET request and errors as they are the easiests one

First define the global offset, log_terminator, and use the global offset that was defined in the previous steps

```
------------------------------------------------------------------------
| int globalOffset = 0;                                                |
| //This is the terminator that every thread will use for logging      |
| std::string log_term = "========\n";                                 |
```

```
|                                                                    |
| //For GET request                                                  |
| //define a std::string that will combine the command with the name |
| //of the file                                                      |
| //"GET FILENAME" will be the placeholder for the log command       |
| std::string logCommand = "GET FILENAME length 0\n"                 |
| //Get the size of the logCommand and log_term for predefining size |
| //of buffer for sprintf()                                          |
| int logSize = strlen(log_term + logCommand);                       |
| //Now define the buffer for sprintf()                              |
| char logGet[logSize];                                              |
| sprintf(logGet, "%s", (logCommand + log_term));                    |
|                                                                    |
| //Now write the buffer to the fd of the log file                   |
| pthread_mutex_lock(&globalOffset_mutex);                           |
| pwrite(logFD, logGet, logSize, globalOffset);                      |
| globalOffset += logSize;                                           |
| pthread_mutex_unlock(&globalOffset_mutex);                         |
|                                                                    |
| //For Errors                                                       |
| //Where %d is the number of the error                              |
| std::string logHeader = "FAIL:.."+logCommand+"..response %d\\n\n"  |
| int logSize = strlen(logHeader.c_str());                           |
| char logGet[logSize];                                              |
| sprintf(logGet, %s, logHeader.c_str());                            |
|                                                                    |
| pthread_mutex_lock(&globalOffset_mutex);                           |
| pwrite(logFD, logGet, logSize, globalOffset);                      |
| globalOffset += logSize;                                           |
| pthread_mutex_unlock(&globalOffset_mutex);                         |
----------------------------------------------------------------------
```

Now that the design for logging GET and any error messages is complete the algorithm for logging PUT must be made

Logging put is a little bit difficult as an algorithm needs to be created for indicating how much space the content is going to be as the content for the log will be the characters from the data converted into hex followed by the count of characters padded by at most 8 zeroes for every line

Firsttly, each full lines will contain 69 characters and the last line will contain how much data is left.

To calculate this, the number of full lines will be the content length of the data being retrieved divded by 20. The last line will be the content length mod 20.

Now that the number of full lines has been retrieved and the number of characters left for the last line has been retrieved, the following

equation can be used to calculate the space that will be allocated to the data.

```
| int full = contentLength/20;                                          |
| int remaining = contentLength % 20;                                   |
| int data_space = (full*(8+(20*3)+1))+(8+(remaining*3)+1);             |
```

Now that the data space has been retrieved, it will be added with the header containing the command, name of file, and length of the file.

Just like with logging GET, a buffer of size data_space+logheader+term will be created.

After creating the buffer, once the reply PUT function is reading in the data using read(), there will be a loop going through the buffer that read() produced and sprintf will convert every character in that buffer to a hex value and put it into sprintf().

Additionally, for every 20 characters, 8 padded zeroes with the number of characters from the last line will be added to sprintf()

```
| //Outside of loop of read(), first delcaration                        |
| char *logPutPointer = &logPut[0];                                     |
| logPutPtr += sprintf(logPutPtr, "%s%08d", logH.c_str(), buff_pos);    |
| //Inside of loop for read()                                           |
| loop {                                                                |
|    for(;pos < bytes_read; pos++) {                                    |
|       logPutPtr+=sprintf(logPutPtr,"%02x",buffer[pos] & 0xff);        |
|       if((pos % 20) == 19) {                                          |
|          logPutPtr += sprintf(logPutPtr, "\n%08d", pos+1);            |
|       }                                                               |
|    }                                                                  |
| }                                                                     |
|                                                                       |
| //After read() loop concluded                                         |
| sprintf(logPutPtr, "\n%s\n", log_terminator.c_str());                 |
| pthread_mutex_lock(&globalOffset_mutex);                              |
| pwrite(logFileFD, logPut, logSpace, globalOffset);                    |
| globalOffset += logSpace;                                             |
| pthread_mutex_unlock(&globalOffset_mutex);                            |
```

Now that the logging for PUT has been finished, alongside with the logging for GET and error messages, this concludes the logging portion for the program.

Furthermore with the algorithm for multi-threading completed, this

concludes the design of Assignment 2