Assignment 3: Adding Aliases to HTTP Server Design Doc Emil Ingemansson (eingeman)

Goals:

The goal of this assignment is to allow aliases to be used within our HTTP server. They should be implemented through the use of a hash table and aliases can point to other aliases as long as they resolve within 10 deep. This means we need to update out performRequest function which contains what to do on gets and puts because we need these functions to search the hash table for the address if we receive an alias in our get or put.

Design:

(Sections in **bold underline** are what I inserted in assignment three to update server to meet the specs)

1) PerformRequest(){
    1) If the request is a GET
        a. **Parts b and c are in a loop that goes through a maximum of 10 iterations. (Incase there is a circular reference.**
        b. **Check the hash table for a key that has the same value as "address" if our address is not a 27 character alphanumeric value because then it must be an alias.**
        c. **If the bucket at hash(key) doesn't have the same key, we need to probe to the next buckets (+1) until we either reach the bucket with our key or reach an empty bucket. If we reach an empty bucket first, that means that our "key" isn't in the hash table and we need to return error 404 not found.**
            i. **If we reach the bucket with our "key" then we can set the address to the value at that key. We can now go back to the start of the loop. If the alias points to another alias, it won't be 27 alphanumeric characters - or _ and thus will restart the loop.**
        d. First, our server will try to open the file in read only with resource name that was taken from the HTTP Header.
            i. If this read returns -1, then the file does not exist and we can return our http header with status code 404 Not Found.
            ii. Else the file exists and we can perform the get for the client.
        e. If step 5a is successful, first we need to use fstat() to get the size of our file using the file descriptor we opened.
        f. Once we know the size of the file, we can send back our HTTP header to the client containing our status code and content length.
        g. After we send the header, we can then read from the file we opened and write the bytes we read back to the client.

h. ONCE WE ARE DONE WE CAN LOG THE FILE BY CALLING FUNCTION LOGFILE() SINCE THE FILE HAS BEEN CREATED IN OUR SERVER'S DIRECTORY AND CLOSED.

2) Else if the client's request is a PUT
   a. **Parts b and c are in a loop that goes through a maximum of 10 iterations. (Incase there is a circular reference.**
   b. **Check the hash table for a key that has the same value as "address" if our address is not a 27 character alphanumeric value because then it must be an alias.**
   c. **If the bucket at hash(key) doesn't have the same key, we need to probe to the next buckets (+1) until we either reach the bucket with our key or reach an empty bucket. If we reach an empty bucket first, that means that our "key" isn't in the hash table and we need to return error 404 not found.**
      i. **If we reach the bucket with our "key" then we can set the address to the value at that key. We can now go back to the start of the loop. If the alias points to another alias, it won't be 27 alphanumeric characters - or _ and thus will restart the loop.**
   d. First, the server will try to open the file and if that returns a nonnegative value, we know that the file already exists.
      i. If the file already exists, we will want to overwrite it so the server will just delete the file that already on the server and create a new file with the same address with what the client is sending. This can be done by specifying the O_TRUNC value in open which will truncate the value to zero when opening a file if it already exists.
   e. Then, we call open to make a new file on the server at the address (with the name) specified by the client with flag O_CREAT and O_RDWR.
      i. Here we can send back the address because we know we can read the bytes in. We will send back a 201 Created HTTP header because we will be making a new file in the server with the address specified by the user.
      ii. We can then we read in the data from the content that the client sent.
   f. ONCE WE ARE DONE WE CAN LOG THE FILE BY CALLING FUNCTION LOGFILE() SINCE THE FILE HAS BEEN CREATED IN OUR SERVER'S DIRECTORY AND CLOSED.
3) **Else if the clients request is a patch**
   a. **First, we will need to read in the data inside the patch request. The first argument of the patch will always be word ALIAS followed by the existing_name and the new_name. We can extract this by reading the content length amount of data and manually assigning the existing name and new name to variables inside the patch function.**
   b. **Once we read in the content length amount of data and have the existing name and the new name, we can enter the new name as the key and the existing name as the value.**

i. **First, we will lock the table so another thread doesn't try to write to the hash table at the same time.**
ii. **Then we will use our hash function to and put our alias(key) and the address its pointing to (value) at the value it hashes to.**
iii. **We will pwrite into the file at offset 128 * (number key hashed to) + magic number, so we don't interfere with another bucket.**
1. **If this bucket has already been filled, then we will store our data at the next +1 bucket (next 128 bits). We will keep +1ing until we find an open bucket.**

c. **Once we are done, we can log the body of the patch request. "Alias *new_name existing_name*"**

2) In main function

a) The server in assignment 2 will have two additional optional flags. The -N flag followed by a number will specify how many "worker" threads the http server will have. Second, the -L flag followed by a filename will specify the server to create a log and to dump the contents along with request, address and content length a file with that name. **Also, if the -a option is specified, then we will want to call open with the name parameter specified. We will have the O_CREAT flag specified so if the file does not exist we will create it but not the O_TRUNC flag in case the file already exists and we do not want to override it.**

b) If we created the file, we can append our magic number to the beginning to verify that it's a valid hash file if our server runs again. Then, we can allocate the rest of the space needed in the file.

i) If the file already exists, we need to check the magic number at the beginning to make sure the file is valid.

ii) If invalid, then we can exit with an internal error because we've been passed a hash table file which is incompatible. (Does not have the same magic number).

3) In ParseHeader

a) Had to change the beginning of the parseHeader function because I made an assumption that the first line would always be 31 characters long excluding the HTTP/1.1.

i) To fix this, I implemented a similar technique to what I did for scanning the content for scanning the first line because Sometimes the header could be a lot shorter and we need to stop when we reach a two spaces. (Meaning we got a request (first string) and some sort of address whether alias or not (second string with space afterwards)

b) Rest of parseHeader function untouched from Assignment 1.

4) Logging

a) **I will create a separate log function that will handle the patch requests. The function will just take the strings I create in the patch part of perform request and write it to the appropriate location in the log file while "allocating" the space for it.**