Design Document: Assignment 3 Adding Aliases to the Web Server

Ryan McCrory
CruzID: rmccrory

1 Goals:

The goal of this assignment is to modify assignment 2 so that it supports aliases. An alias is an alternate name that can be used to refer to an object stored in the HTTP server's hash table. This will be done by implementing a version of the PATCH command that will add a name as an alias for an existing object.

2 Design:

There will be several parts to the design. First, I will do the command line parsing in the main method, where I will use getopt() to determine the options and to name the IP address and port number. Next, I will open a network socket on the server. Then, I will create the number of worker threads specified by the user, or 4 by default. I will have a dispatcher thread signal worker threads to execute commands from incoming connections. Workers will sleep when there is no work to do, and the dispatcher will sleep when all the workers are busy. Finally, the client sends a request to the server that either asks to send a file from client to server (PUT) or fetch a file from server to client (GET). This is processed in a worker thread. If the -l flag is used, this will be logged in a file named by the argument following -l.

I will create or modify a mapping file if the -a flag is used. If it is used and the file already exists, I will edit the file using pwrite. I will create a separate function that will handle PATCH requests. In my parse function, I will check the first word of a request header and if it is 'PATCH', I will call the patch() function similarly to how the get and put functions are called. This function will write the resource name and alias to the mapping file, which will be accessed via the hash function from city hash.

2.1 Global Variables

In order for the threads to interact with one another, I am going to use several global variables. There is a bool to determine whether there is logging or not, a queue which will hold all the socket file descriptors, and an int which will keep track of the number of available threads. Then, there are locks and condition variables. I am using several locks, one to lock the queue while a thread reads or writes to it, one to sleep a worker, one to sleep the dispatcher, and one to lock the available threads int. There will also be a file descriptor associated with the log file.

There will also be a couple global variables to deal with mapping. One is a bool to determine if there is or is not mapping. Another is the file descriptor associated with the mapping file. There is also an int to determine where the location of the new alias is going in the mapping file. There is also an array to hold up to 8000 addresses of aliases in the mapping file.

```
//global variables
Int address [8000];               //array to hold up to 8000 addresses of aliases
bool is_log;                          //determine if logging needs to be done
Bool is_mapping;                      //determine if mapping needs to be done
std::queue<int> clients;          //a queue that holds the socket file descriptors
int available_threads;        //keeps track of how many threads are available to work
Int maping_file;                              //file descriptor for the mapping file
Int start_offset;       //the location of the first new place to write in mapping file
Int log_file;                                         //file descriptor for the log file

//locks and condition variables
pthread_mutex_t mutex_que = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_thread = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_dispatch = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_avail = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_thread = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_dispatch = PTHREAD_COND_INITIALIZER;
```

2.2 MAIN:

The main method begins by parsing the command line arguments. I am going to use getopt() to do this, since there are going to be optional flags for this program. Using getopt, I will set a variable to the hostname and port number (if port number is not specified, it is set to 80 by default). If the -N flag is used, I will set a variable num_threads to optind (the arg immediately after the flag), and if -N is not used, I will set num_threads to 4 by default. If the -l flag is used, I will set the global variable is_log to true, or set it to false if not. If it is used, I will also set a variable to the arg after the flag, and create the log file with the name of that arg. If the -a flag is used, the name associated with it will be the name of the mapping file, which will hold the aliases. Then, I will open a socket using the code given by professor miller. Once the socket is opened, we accept the connection from the client with an infinite loop, and then wait for a get or put request.

Next we deal with mapping. If there is no mapping, the server exits according the the assignment spec. If there is, we either create a file with the name specified by the -a flag, or, if it already exists, we open it for editing. Then we initialize our variables, which include an int array to hold the locations of the resource names and aliases in the mapping file, a buffer, a counter, and the start_offset which starts at 1. We then iterate through the mapping file and use pread at every 128 bytes to determine where the next available increment of 128 bytes is. We then set start_offset to this number. While we do this iterating, each time we find an existing alias in the mapping file, we hash it using city hash and mod this number by 8000, since there are 8000 entries in the array. Then we set the index at this number of the address array equal to offset_start. This will be used later to access the proper location of the mapping file.

Then we enter an infinite loop, where we accept connections from clients. We then push the clients file descriptor to the global queue. Then, we enter a while loop that runs while available_threads is equal to 0, meaning all worker threads are busy. In this loop, the dispatcher thread goes to sleep and waits for the signal from a worker to tell it that is is available. When this happens, available_threads is incremented so we exit the loop. Then, we signal the worker thread to begin to work on the clients request, and we go back to the start of the infinite loop, listening for another connection. Each time we use shared memory here, we must lock the associated mutex before, and unlock it after.

```
Input: argc, argv[]

Int main (){
        Use getopt for “lN”
        Set hostname and port number
        If (-a flag){
                Is_log = true;
                Mapping_name = optarg;
        }
        If (-l flag){
                Create log file       //here we create the log file that we will later write to
        }
        If (-N flag){
                Num_threads = optind;
        } else {
                Num_threads = 4;
        }
        If (is_mapping == false) {
                exit(0);                //exit is no mapping, according to asg specs
        Set up sockets        //code given by professor miller, I'm not writing it here
        listen(main_socket, 16);
        // deal with mapping if -a is used
        if (is_mapping == true){
                mapping_file = open(mapping_name, O_CREAT | O_RDWR, 0644);
                //start_offset at 1 bc if hash returns 0, it is an invalid alias
                start_offset = 1;
                //allocate memory for char *’s
                char *buff = (char *) malloc(1024 * sizeof(char));
                char *alias = (char *) malloc(1024 * sizeof(char));
                char *resource = (char *) malloc(1024 * sizeof(char));
                int read_bytes = pread(mapping_file, buff, 128, start_offset);
                //read until we find end of mapping file
                while (read_bytes > 0){
                        sscanf(buff, "%s %s \n", alias, resource);        //parse buffer
                        //place offset into array
                        address[CityHash32(alias, strlen(alias))%8000] = start_offset;
                        start_offset += 128;          //increment offset by 128
                read_bytes = pread(mapping_file, buff, 128, start_offset);
            }
        }
        For (i=0; i<num_threads; i++){
            create(&workers[i], NULL, handle_connection, NULL);
        }
        while(true){
            int32_t cl = accept(main_socket, NULL, NULL);
            lock(&mutex_que);                              //critical region start
            clients.push(cl);
            unlock(&mutex_que);                            //critical region end
            lock(&mutex_avail);
            while(available_threads == 0){     //sleep while no available threads
                unlock(&mutex_avail);
                wait(&cond_dispatch, &mutex_dispatch);
                lock(&mutex_avail);
            }
            unlock(&mutex_avail);
            signal(&cond_thread);              //wake a worker thread to start work
        }
    }
```

Algorithm 1: MAIN

2.3 Handle Connections:

I will begin to implement multithreading in my main method. There, I will create N new threads, where N is specified by the -N flag, or 4 by default. Those threads will be sent to this function, where they will immediately enter an infinite loop and go to sleep. Then, when a thread receives the signal to wake up from the dispatcher thread, it will. It will then use the mutex, mutex_avail, to lock available_threads, and decrement it, since this thread is about to start doing work. We will unlock mutex_avail, then lock the mutex, mutex_que, and pull the next file descriptor from it and then remove it from the queue, and then unlock mutex_que. Then we call parse with the client file descriptor, where the program continues as asgn1 did, but in this worker thread. When the GET or PUT actions are finished and control returns to this function, we lock mutex_avail again and increment it, since we finished the work we were doing and are available to work again. We then unlock mutex_que and signal the dispatcher thread that we are done, in case it is sleeping. We return to the top of the infinite loop, and go back to sleep while we wait for a connection.

```
Input: pointer to socket

void* handle_connection(void *arg){
    While (true){
        wait(&cond_thread, &mutex_thread);          //wait for a connection
        lock(&mutex_avail);
        Available_threads--; // In critical region, decrement available threads
        unlock(&mutex_avail);
        lock(&mutex_que);
        int32_t client = clients.front();      //In critical region, pull file descriptor
        clients.pop();                              //from queue, then remove it
        unlock(&mutex_que);
        parse(client);                         //Call parse with client file descriptor
        lock(&mutex_avail);
        Available_threads++;   // In critical region, increment available threads
        unlock(&mutex_avail);
        signal(&cond_dispatch);                     //alert dispatcher when finished
    }
}
```

Algorithm 2: handle_connections

2.4 Parse:

We call the parse function with the client file descriptor we got from accept(). We receive a header from the client and we parse it using sscanf() to determine if it is a GET or PUT request, and to determine the name of the file from the client, and the name of the file on the server. Next we make sure this is a valid request header, so we check if the resource is 27 valid ascii characters, or an alias. If it is 28 characters, but the first character is a ‘/’, we remove the ‘/’ and update the resource name. We then call either the get() , put(), or patch() function depending on what we determine the client to be doing. The parameters we pass get() and put() are the client file descriptor, the file name, and the number of bytes we are reading, which is 1024.

Input: client file descriptor

```
parse(int cl){
    //read incoming data
    Int bytes = 1024;
    Char *buffer = (char *) malloc(bytes * sizeof(char));
    read(cl, buffer, bytes);
    //declare variables
    Char *action = (char *) malloc(bytes * sizeof(char));
    Char *file_name = (char *) malloc(bytes * sizeof(char));
    //parse data using scanf
    sscanf(buffer, “%s %s \n”, action, file_name);
    logging(fd, action, 1024, 0);             //pass needed info to logging() function
    logging(fd, file_name, 1024, 1);
    //error checking
    If (file_name has 28 chars){
        Str temp = file_name;
        If (temp.at(0) = ‘/’){
            temp.erase(first char);
            File_name = temp;
        }
    }
    //call put or get depending on what action* is
    If (action is “PUT”){
        put(cl, file_name, bytes);
    Else if (action is “GET”){
        get(cl, file_name, bytes);
    } else if (action is "PATCH"){
        patch(cl, file_name, bytes);
    } else {
        send(400 Bad Request error);
    }
    //free memory
    free(buffer);
    free(action);
    free(file_name);
}
```

Algorithm 3: Parse

2.5: PATCH:

The patch function will begin by initializing all the char * variables it will be using. This includes a buffer to read the incoming data from the client, an action variable which will be set to PATCH given that we are in the PATCH function, a resource variable which will be set to the existing name sent by patch, a alias variable which will be set to the new name sent by PATCH, and an entry variable which will include both the alias and the resource separated by a space. After we malloc these, we receive the body from the PATCH request, and parse it as described above. We then determine if it is valid by making sure the resource name and alias are not over 128 bytes. If they are, we reply to the client with a 400 Bad Request message. If it is ok, we combine the alias and the resource name in entry, and write that to the mapping file using pwrite. We will write at the location of the next available 128 byte increment determined in main with the start_offset variable. Then, we will reply to the client with a 200 Ok message. Lastly, we will free our memory.

Input: client file descriptor, file name, a number of bytes (1024)

```
void patch(int32_t cl, char *file_name, uint32_t bytes){
    //declare variables
    char *buffer = (char *) malloc(bytes * sizeof(char));
    char *action = (char *) malloc(bytes * sizeof(char));
    char *resource = (char *) malloc(bytes * sizeof(char));
    char *alias = (char *) malloc(bytes * sizeof(char));
    char *entry = (char *) malloc(bytes * sizeof(char));
    recv(cl, buffer, bytes, 0);                              //receive data from client
    sscanf(buffer, "%s %s %s \n", action, resource, alias);           //parse data
    if ((strlen(resource) + strlen(alias)) > 128 ){
        send(cl, 400 Bad Request header 0);     //send bad request response
    } else {
        //write to mapping file starting at next available increment of 128
        sprintf(entry, "%s %s\n", alias, resource);    //combine alias and resource
        pwrite(mapping_file, entry, 128, start_offset);
        address[CityHash32(alias, 128)%8000] = start_offset;
        send(cl, 200 ok header, 0);                 //send ok header
    }
    free memory
```

}

2.6: PUT

When there is a put request, parse will call this function. First, we check if there is mapping in this run of the server. If there is not, we proceed to the next paragraph. If there is, I will set an int called spot equal to the value in the address array at the position returned by CityHash32(file_name, 128)%8000. This is where the alias and resource name for the given alias is. Then we allocate memory for the char *'s resource, alias, and buff. We use pread to get the 128 byte bucket starting at spot, and then parse it to isolate the alias and resource name. Then, we set file_name equal to resource and continue as if there was no mapping from there.

Next, we will use a similar algorithm to asgn0, which will read the contents of the file we are trying to put. Instead of writing it to std out like in asgn0, we will be creating and writing it to a new file, which will have the same name as the request target curl specifies. After we are finished writing the file, we send the client a response header, indicating if this was successful or not.

Input: client file descriptor, file name, a number of bytes (1024)

```
put(int cl, char *file_name, int bytes){
    //deal with mapping
    If (strlen(file_name != 27) {
        //set spot equal to file_name's offset in mapping file
        int spot = address[CityHash32(file_name, strlen(alias))%8000];
        //allocate memory for char *'s
        char *resource = (char *) malloc(bytes * sizeof(char));
        char *alias = (char *) malloc(bytes * sizeof(char));
        char *buff = (char *) malloc(bytes * sizeof(char));
        pread(mapping_file, buff, 128, spot);
        sscanf(buff, "%s %s \n", alias, resource);              //parse
        sprintf(file_name, "%s", resource)  //replace file_name with resource
        if (spot == 0){                     //if spot is 0, alias doesnt exist
            //send bad request response
            send(cl, 400 Bad Request header 0);
        }
    }
    Char *buffer = (char *) malloc(bytes * sizeof(char));
```

```
        Int read_bytes = recv(cl, buffer, bytes, 0);
        //create the new file with the proper permissions
        Int new_file = open(file_name, O_CREAT | O_WRONLY,  0644);
        //write to the new file, same code as dog() in asgn0
        While (read_bytes > 0){
                write(new_file, buffer, read_bytes);
                Read_bytes = recv(cl, buffer, bytes, 0);
        }
        //send response header to client
        If (successful) {
                Send (successful header message);
        } else {
                Send (unsuccessful header message);
        }
        //close file and free memory
        close(new_name);
        free(buffer);
}
```

Algorithm 4: PUT

2.7: GET

When there is a get request, parse will call this function. It is also similar to dog() from asgn0. First, we check if there is mapping in this run of the server. If there is not, we proceed to the next paragraph. If there is, I will set an int called spot equal to the value in the address array at the position returned by CityHash32(file_name, 128)%8000. This is where the alias and resource name for the given alias is. Then we allocate memory for the char *'s resource, alias, and buff. We use pread to get the 128 byte bucket starting at spot, and then parse it to isolate the alias and resource name. Then, we set file_name equal to resource and continue as if there was no mapping from there.

Then, we check to see if the file the client is trying to get exists on the server side. We send the client a response header letting it know if it does or not, and the content length of this file. Then, if it exists, we open the local file on the server side that the client is trying to get. Then we read this file and place its contents in a buffer, which we send to the client. This is done in a loop that runs until all bytes of the file have been read and sent.

Input: client file descriptor, file name, a number of bytes (1024)

```
get(int cl, char *file_name, int bytes){
    //deal with mapping
    If (strlen(file_name) != 27) {
        //set spot equal to file_name's offset in mapping file
        int spot = address[CityHash32(file_name, strlen(alias))%8000];
        //allocate memory for char *'s
        char *resource = (char *) malloc(bytes * sizeof(char));
        char *alias = (char *) malloc(bytes * sizeof(char));
        char *buff = (char *) malloc(bytes * sizeof(char));
        pread(mapping_file, buff, 128, spot);
        sscanf(buff, "%s %s \n", alias, resource);          //parse
        sprintf(file_name, "%s", resource)  //replace file_name with resource
        if (spot == 0){                     //if spot is 0, alias doesnt exist
            //send bad request response
            send(cl, 400 Bad Request header 0);
        }
    }
    //open file on server side and send response
    int local_file = open(file_name, O_RDWR);
    if (local_file == -1){
        warn("%s", file_name);
        send(response header indicating fail);
    }
    send(response header indicating success);
    //read and write data, similar to dog() from asgn0
    char *buffer = (char *) malloc(bytes * sizeof(char));
    int read_bytes = read(local_file, buffer, bytes);
    while (read_bytes > 0){
        send(cl, buffer, read_bytes, 0);
        read_bytes = read(local_file, buffer, bytes);
    }
    //free memory
    free(buffer);
}
```

Algorithm 5: GET

Note: Below is not finished

2.8: LOGGING

There is a global bool called is_log that is initialized to false, but if the -l flag is specified, it is set to true. When it is true, a logging file will be created in main, and written to in parse as well as get and put. When necessary, those functions will call this function, which will to the actual writing to the log file. They will pass this function the file descriptor to write to, the buffer with the data to write, the size of the data, and the offset which will allow us to know where in the log file to write specifically.

Note: I know this doesn't work, but I focussed my time on the spec for assignment 3 rather than fixing this.

Input: fd, buffer, size, offset

```
Logging (fd, buffer, size, offset){
        pwrite(fd, buffer, size, offset);
}
```
