

Design Document:

Adding Aliases to the Web Server

Anthony Ho
CruzID: Ankho

1) Goals

The goal of this assignment is to modify our HTTP server from assignment 2 to support aliases. The web server should not support PATCH commands which will add a name as an alias for an existing object.

2) Design

The design will include 5 parts. When the program initializes, we will have a global queue which will contain connection requests as they come in, a boolean which will indicate if logging is enabled, a log offset variable, and a variable to contain the log file. We will then parse the command line arguments so that we know how many threads the server should have and if they want logging. Then we will create the correct number of threads (user inputted or 4 by default) and have them wait. Then we will open the socket and start filling the queue with requests. When a thread takes a requests from the queue, they will then begin parsing the requests and send it to either the GET, PUT, or PATCH functions. The last part will be a function that handles logging requests to a log file if the user wants logging.

2.1) Parsing Command Line Arguments

Parse the command line arguments in order to determine what flags are provided. (How many threads, logging) This can be done with the following pseudo code:

Input: Argument count: argc

Input: Array of arguments: argv

If argc >= 3:

 For i = 1 to argc :

 If argv[i] == "-N":

 numThreads = argv[i+1]

 i++

```

If argv[i] == "-l":
    logFile = argv[i+1]
    i++
If argv[i] == "-a":
    aliasFile = argv[i+1]
    If file argv[i+1] doesn't exist:
        Create file argv[i+1]
    Else if the file exists:
        If file does not contain the magic number:
            Exit with error "Invalid mapping file"
    i++
Else:
    Hostname = argv[i]
    If i != argc:
        Port == argv[i+1]
        i++
    Else:
        Port == default port 80

```

2.2) Create all threads

Create the number of user inputted threads (Or 4 by default.) Then put all the threads to sleep while waiting for a connection. Like so:

Allocate space for the threads

For 0 to number of threads:

 Call threadCreator

Def threadCreator:

Input: void * arguments

while(1):

 Lock queue of requests

 If queue is empty:

 wait();

 If queue is not empty:

 Socket = dequeue();

 Unlock queue of requests

 Pass socket to parser function

Close the socket

2.3) Opening a socket

Open a network socket by including and using the functions in `sys/socket.h` like so:

Input: Argument count: `argc`

Input: Array of arguments: `argv`

Initialize `hostname` and `port` as `argv[1]` and `argv[2]` (The command line arguments)

Initialize a main socket using `socket()`

Use `setsockopt()`

Use `bind()` to bind the socket to the `hostname` and `port`

Use `listen()` to put our socket into a passive state and wait for a client

`while(1):`

`new_socket = accept()`

 Create a buffer of size 16KiB

 Lock queue of requests

 Enqueue the request

 Signal the queue is not empty

 Unlock the queue of requests

2.4) Parsing the Request

After a thread takes a connection off the queue, it will begin to parse the request and pass it to either the GET or PUT functions to be executed.

Input: `socket`

Use `strtok()` to split the string using the delimiter of a space (first two tokens will be request and the resource name)

Create a pointer to the "Content-Length:" header using `strstr()`

Create a variable to store the content length

If the content-length header exists:

Add 16 to the pointer (to get to the number)

`while(digit remain):`

If current char is a digit:

Add current digit to content length variable

Increment pointer

Create variables for "GET" and "PUT"

Compare the first string after splitting with the variables to determine if its a GET or PUT

Pass the socket and the resource name to the respective function

2.5) GET vs PUT vs PATCH

Depending on the request type, the server will either send the contents from a file on the server to the client or store a file from the client onto the server. As of Asgn3, we will also include support for PATCH requests which will add an alias for a resource. We will also have to add an additional function to find the resourceName if an alias is provided. This will be handled by two different functions that are defined as so:

2.5.1) GET

The get function will take the 27 character resource name and find that file and send the contents to the client with a content-length and status code 200 OK. If the file does not exist on the server, it will return status code 404 file not found. This will also check the validity of the resource name. This will be done with the following algorithm:

Input: socket

Input: char* resName

Initialize an array that contains all the valid characters for a resName. (A-Z, 0-9, -, _)

Check if the resource name is the correct length (27) and contains proper chars

^ Else return status code 400 through the socket

Use open() to find the file on the server

if(file exists and cannot be accessed):

 Write status code 403 through the socket

Else

 Initialize a variable to track content length

 if(file exists) :

 Loop through file to get the length of the entire file

 Create a buffer of size 32KiB

 Write status code 200 through the socket

 Write content length through the socket

```
        While there is still content to be written:
            Read the file
            Write the content through the socket using write()
    Else if file doesn't exist:
        Write status code 404 through the socket
```

2.5.2) PUT

The put function will start similarly to the get function as it will check the resource name for any errors. The function will then attempt to save the file from the client into the server. If a content length is provided, it will only store up to the content length, otherwise it will save the entire file. This function will send a status code 200 OK if it is able to save the content into a file on the server. It will send a status code 201 File Created if the file did not previously exist and it creates the file.

```
Input: socket
Input: char* resName
Input: int contentLength
Initialize an array that contains all the valid characters for a resName. (A-Z, 0-9, -, _)
Check if the resource name is the correct length (27) and contains proper chars
^ Else return status code 400 through the socket
if(file exists and cannot be accessed):
    Write status code 403 through the socket
Use open() to check if the file already exists in the server
If the file does not exist:
    Use Open() to create a file with read and write permissions
    If a content length was provided:
        While content length isn't reached:
            Read() content from the socket
            Write() the content to the file you created
            Update content length with new value
    Else
        While client doesn't terminate connection:
            Read content from the socket
            Write the content to the file you created
        If client terminates connection:
            stop
```

```
    Write status code 200 through the socket
Else (file already exists)
    Use open() to overwrite the file
    If a content length was provided:
        While content length isn't reached:
            Read() content from the socket
            Write() the content to the file you created
            Update content length with new value
    Else
        While client doesn't terminate connection:
            Read content from the socket
            Write the content to the file you created
            If client terminates connection:
                Write status code 201 through the socket
```

2.5.3) PATCH

The patch function will take in an alias name and a target name. Then the function will hash the alias name and reduce the hash to a number between 1-8000. This number will be used as an index in the hash table. Then we will open the alias file and add an entry for the alias with the target name at the index specified. Also, whenever we write or read from the mapping file, we need to make sure to lock the file so that no other thread will access it at the same time. This will prevent issues where a thread attempts to write at the same index as another thread is trying to read/write from it. This will be done with the following algorithm:

```
Input: socket
Read from the socket to get aliasName and targetName
Check if aliasName contains only printable characters
If length of aliasName + targetName > 126 or contains unprintable characters:
    Write status code 400 through the socket
Else:
    If the target is a valid 27 character resource name:
        If file does not exists on server:
            Write status code 404 through the socket
        Else:
            Set offset to hash(aliasName) * 128 + length of magic number
            Lock mapping file
            Write the aliasName and targetName at the offset
```

```

        Unlock mapping file
        Write status code 200 through the socket
    Else:
        Set offset to hash(targetName) * 128 + length of magic number
        Lock mapping file
        Read the key at the offset
        Unlock mapping file
        If the offset is null: (The target alias does not exist in the mapping file)
            Write status code 404 through the socket
        Else: (The target alias exists in the mapping file)
            Set offset to hash(aliasName) * 128 + length of magic number
            Lock mapping file
            Write the aliasName and targetName at the offset
            Unlock mapping file
            Write status code 200 through the socket

```

2.5.3.1) Hashing

For this program, I will be using the djb2 hashing algorithm that can be found on <http://www.cse.yorku.ca/~oz/hash.html>. I modified this algorithm by modding the final value by 8000 to reduce the has to a value between 1-8000.

2.5.4) findResource

This function will look through the mapping file to find the resource name in order to pass it to the GET and PUT functions. This will be done by the following algorithm:

```

Input: char* name, length, request
Input: int socket
If name is a valid 27 character resourceName:
    If request == GET:
        handleGet(socket,name)
    Else if request == PUT:
        handlePut(socket,name,length)
Else: (name is an alias)
    Set offset to hash(name) * 128 + length of magic number
    If value at the index is null: (Alias does not exist)
        Write status code 404 through the socket
    Else: (Alias does exist)
        Get the targetName from the mapping file

```

```
        findResource(socket,targetName,length,request)
        // This is a recursive call to the function as we will keep jumping through
aliases until a valid resource name is found.
```

2.6) Logging

Logging will be called through the get and put functions. If the request fails, it will log the request and the response code. If the request succeeds, it will log the request, with the length, and the data being sent in hex.

2.6.1) Logging the header

For logging the header, we will first check if the response was successful. If the request failed, it will move the offset in the logfile to reserve space and then write the fail message. If the response was successful, it will move the offset to reserve space for the header and the data and write the header and data into the log file. Shown here:

Input: char* request, resname, length, response

Input: int file

Input: bool success

```
originalOffset = offset;
```

```
if(success):
```

```
    Lock offset variable
```

```
    Offset = offset + (length of "FAIL: + request + resname + HTTP/1.1 + ---
response + response + \n") + length
```

```
    Store this offset variable in a separate variable
```

```
    Offset = offset + length
```

```
    Unlock offset variable
```

```
    Pwrite the header from the original Offset
```

```
    Call function to log data and pass it the offset variable we stored earlier
```

```
Else:
```

```
    Lock offset variable
```

```
    Offset = offset + length of "FAIL: + request + resname + HTTP/1.1 + --- response
+ response + \n"
```

```
    Unlock offset variable
```

```
    Pwrite the header from the original Offset
```

2.6.2) Logging the data

After logging the header, we will then log the data from the file using the offset passed from the header function. The log of data will be written in hex, 20 bytes per row. The algorithm for converting and logging is as follows:

Input: int dataOffset

Input: int file

Create a buffer to store every 20 bytes of data

While there is still data in the file:

- Create a buffer for the left column

- Create a buffer for the data

- Write the left most column (zero padded byte count)

- Increase offset by 8

- For 0 to the size of data:

 - Create a buffer for a single hex byte

 - Convert a character to hex with a space in front (divide the char by 16 for first hex digit, and mod by 16 for second hex digit)

 - Add the new hex value to the data buffer

- Write the row of data to the file

- Increase offset by the length of the data

- Write a new line to the file

Write =====\n to the file