Design

Overview:

The design is broken up into header files and the main HTTP proxy file as described below including the functions in each file:

- Httpserver.c
  - Dispatcher
  - HealthProber
  - Worker_threads
  - main
- RequestHanlder.h
  - strouint16
  - Create_listen_socket
  - Create_client_socket
  - checkRequest
  - Handle_connection
  - check_health
- DataStructs.h
  - createCacheEntry
  - createCache
  - insertFileToCach
  - delCache
  - findFileInCache
  - CreateServersListEntry
  - insertServersList
  - succesRate
  - Best_server
  - findPort
  - CreateSharedData

These functions make up the HTTP proxy program. I created separate functions for each request because this will make the code more legible to read and create a more organized code. Below I will describe the function and the way I designed them.

DataStucts.h

This file has all the function and declaration for the data structures used in this program. The following data structs are declared in this file:

```
typedef struct Cache{
  struct tm* date;
  char* fileName;
  char* data;
  int file_size;
  struct Cache *next;
  struct Cache *prev;
}Cache;

typedef struct ServersList{
  int requests, fails, disable;
  u_int16_t port;
  struct ServersList *next;
  struct ServersList *prev;
}ServersList;


typedef struct Shared{
  int* queue;
  int start, end, size, cache_size, cache_length, healthcheck, total;
  ServersList* servers;
  Cache* cached_files;
  sem_t full, empty, lock, healthlock, check;
}SharedObj;
typedef struct Shared* Shared;
```

- The Cache and ServersList are both linked lists.

- For the Cache data struct it is pretty self explanatory what data it hotels more. The data array holds the cached file

- For the ServersList, requests and fail are the counters for load balancing. The disable integer is used to keep track of server that are down, if set to 1 it indicates it is disabled.

- The Shared data Structure is the data that is used by the threads to communicate with each other. Both ServerList and Cache are also in Shared data Structure

createCacheEntry function

- Takes in a size for the cache size
- Allocates memory for filename, date and data which holds the file contents
- Return the cache

insertCache function

- Inserts a chase entry at the head, takes in the head and a new cache datastuct
- If nothing at the head just set the new cache at the head
- Else move the head to previous of the new chase and so the new entry is at the head

createCache function

- This function takes in the cache size and amount of caches, returns the head of the new cache created
- Initializes the head of the cache

- Loops till the amount of caches request are created and inserted into the list using createCacheEntry and insertCache
- Return the head of the linked list

insertFileToCache function

- Takes a cache entry c, filename, tm structure date, and size for file size
- Sets the values given to the fields for c

delCache function

- Deletes the given cache del form the linked list cache head
- If the del it is at the head set the next cache del to the head
- Else
    - Set del's previous' next to del's next
    - If del's next is not NULL set del's next's previous to del's previous
- Free the allocated memory for the field of del and free del

findFileInCache

- Takes share data structure and filename, return ptr to the cache that has that file name else returns NULL
- Sets ptr to cached files field
- Loops till ptr is NULL
    - If filename is the same as ptr's filename
        - Return ptr
    - Set ptr to ptr's next
- Return NULL

findEmptyCache

- Takes in the Shared data structure, and returns a empty Cache entry
- Sets ptr to the head of cached files list from the shared data structure
- Initializes prev cache ptr
- Loops till the ptr reaches the tail of the list
    - Set prev to ptr
    - Set ptr to ptr's next
- Sets the tail to prev
- Set ptr to prev
- Call delCache with the tail and the cache list to remove the tail
- Create a new cache using createCacheEntry
- Insert the cache in to the cached file list
- Return the new chache

createServersListEntry

- Takes in a port number
- Sets port to given port number
- Sets fails and request and disable to 0
- Return the cache entry

insertServersList

- Takes in the head of the server list and port number
- Creates new server list entry using createServersListEntry
- If head is null set the new server list entry to the head
- Else move the head to previous of the new chase and so the new entry is at the head

successRate

- Takes in requests and fail and calculates the success rate
- Rate = requests - fails / requests
- Returns the rate

Best_server

- Takes in the head of the server list
- Sets ptr to the head
- Initializes the port, number of requests and success_rate(has to be a floating point number) to negative one
- Loops till ptr equals NULL meaning it reached the end of the list
    - If ptr's request is less then num of reqs or number of requests is equal to -1 and server is not disabled
        - Set port to ptr's port and set number requests to ptr's requests
        - Set success rate to success rate of ptr using successRate
    - Else if ptr's request is equal to number of requests and server is not disbabled
        - Calculate the success rate of ptr and if is bigger than the success rate from previous success rate
            - Set port to ptr's port and set number requests to ptr's requests
            - Set success rate to success rate of ptr using successRate
    - Set ptr to ptr's next
- Return port

findPort

- Takes in the head of the serverlist and port number and returns the server entry that has matching port number, return NULL if server entry with that port number is not found
- Sets ptr to the head of the server list
- Loops till ptr equal NULL
    - If ptr's port equals the given port number

        - Return ptr
    - Else
    - Ptr is set to ptr's next
- Return NULL

createSharedData

- Takes in in number of threads, server list, caches size, number of chaces, and number of health checks
- Set the variables
- Also allocates space queue
- Also initializes the semaphores for the following
    - Full will check if the queue is full, set to 0
    - Empty to check if the queue is empty, set to N since it is empty
    - Lock to lock the critical areas
    - Check to call the thread does health checks
    - Healthlock, so process are locked when doing a healthcheck

Httpserver.c

Handles connfd connections coming in and uses dispatcher and worker threads design.This is where the main thread creates the worker threads and handles connfd when they are coming in and dispatches them to the worker threads using a queue to handle each connection file descriptor. It also creates a thread for health checking the given servers. It also creates the needed data structures for the list of servers, empty cache and shared data structure.For locks I used semaphore to increase readability and ease of use.

Dispatcher function

- This function is for the main threads and add new connection fds to the queue. It takes in the shared data structure.
- Checks if the queue isn't empty, if so wait, does this using a semaphore
- Checks if lock is set if not set lock
- Adds the connfd to the queue and increase the tail of the queue
- Post the lock so other threads can access the lock
- And post the full semaphore the increment the lock if the queue is full

HealthProber function

- This function probes the servers with health checks, it takes in the shared data structure.
- Has an infinite loop so that the process does not die.
- Using a semaphore checks if a health check has to be done
- If so loops through the server and checks health on all the servers which is in the shared data structure
- Then it will loops posting the healthlock so that the worker thread can process requests again

Worker_threads function

- This function is where the worker threads pull the new connection fds from the queue. It takes in the shared data structure.
- Set a while loop so that a thread continuously grabbing connections that are waiting in the queue
- Checks if the amount of requests made, it will block if there has to be a health check done on the servers, this is done using a semaphore
- Checks if the queue isn't empty if so wait otherwise check if the lock is open, does this using a semaphore
- If the lock is open, lock it and continue
- Acces the queue and pull the first connection
- Increment the head
- Finds the best server using the best_server function from DataStructs.h
- If port is 0 close connection

- Opens connection with the port number using create_client_socket from RequestHandler.h
- Find the server entry using findPort function from DataStructs.h
- Loops till serverfd does not equal -1
    - Disable server setting disable int to 1
    - Call find port using best_server
    - If port == 0 close connection and post lock
    - Update server entry using findPort
    - Opens connection with the port number using create_client_socket from RequestHandler.h
- Increment amount of request to the server which is stored in the server data structure
- Post the lock so other threads can access the queue
- Post the empty semaphore so that the dispatcher knows when it is empty
- If server entry equals NULL send internal error message
- Else call handle connection with the connection fd pulled from the queue also send the shared data struct within the function and server fd and server etntry

Main function

- Using get opt retrieve argument given
- Create server data structure
- Loops through the port number given in arguments and inserts the port number into the servers list, also does a health check using chec_health from RequestHanlder.h
- Creates the shared data struct for the threads.
- Creates a list of threads, and loops through the list and calls create threads for each threads sending it to the worker thread function.
- Creates healthchecker thread and send it to the healthprober function
- This is the main function where the port is set and listen_socket gets called to start listening for connections.
- After setting up the socket and the socket started listening, there is an infinitely loops listening till it connects with a client and calls dispatcher function where it adds the new connection fd to the queue and if the queue is full wait till it isn't, which handles the request made.
- After adding the connection fd to the queue the code loops till it get another client connection.

RequestHandler.h
This header has all the functions need to handle the requests made, checking caching, and parsing headers sent from the client. Also includes the socket functions given.

Strtouint64 function

- This code was given in the start code from assignment 1.
- It converts a string to a 16 bits integer.
- This is called the main function of the code.
- It is used to convert the given port number into a 16-bit number since to create a listening socket the port has to be in 16-bit format.

Create_listen_socket function

- This code was given in the starter code.
- But get called in the main function to create a listen socket, opening up for clients.
- It uses the 16-bit port number and creates the socket and sets the address and port for the socket
- Returning a listen file descriptor

Create_client_socket function

- This code was given in the starter code
- Sets up a client connection with a given port number
- Returns a listening descriptor

Check_request function

- This helper function checks the field given of the request header
- It takes the following string: request type, file name, http version, host
- Checks if request type is GET else return 501
- Loop through file name and checks if file name has any forbidden characters if so return 400
- Checks if length of file name is less than 19 else return 400
- Checks if http version given is HTTP/1.1 else return 400
- Cheeks if host is not 0 else return 400

CheckModified function

- Check if a tm data structure cached_date is less recent than tm data structure server_date
- Return 1 if not, returns 0 if the cached_date is less recent that server_date

Handle_connection function

- This function handles the client's header parsing and sending requests to the servers and if the right size will cache files
- Allocates memory for the buffer buff
- Using recv it will get the request from the client
- Makes a copy and parses the request using strtok
- Then it calls checkRequest and checks if the request made by client is valid
- If it either code 501 or 400 it will send a response to the client of the error and leaves RequestHandler
- Using findFileInChace which is in DataStructes.h, check if file is in cache, it will return NULL if not
    - If not it will send the request made by the client to the server
    - And recv the response given by the server
    - Send that response to the client
    - It will pull the response type returned by the server and check if it is 200
    - If it is 200
        - It will parse the response from the server and find the length and the last modified date using strtok
        - It will find an empty cache using findEmptyCache from DataStructures.h which will return a pointer to the cacher
        - Then inserts file name time and file length that it pulled from the response into the cache using insertFileToCache from DataStructures.h
        - It finds the index where the response header ends
        - If length given is less or equal to the cache size set in the shared data struct
            - Create the tm data structure and use strptime to take the time given from the response header
            - Initialize i which is a holder of where the cache is going to write at
            - Than iterate through the response of the server till either the cache is filled or it reaches the end of buffer, setting each byte of the buffer to the cache data at index i
            - Increment i every iteration of the loop
            - Calculate if there are any more bytes to be read and set it to bytes
            - If bytes is greater than 0 loop till bytes is 0
                - Recv another batch of data
                - And send it to the client
                - Decrease bytes by the amount of bytes read
                - Than iterate through the buffer and set it to cache data
        - If the size of file is too big for the cache
            - Calculate the bytes that need to be read still
            - If so loop till all the bytes have been received and send it to client
            - Decrease bytes needed to be read

    - If the response from the server was not 200 ok increment server fails counter which is an variable for the server pointer
- If File is in cache
    - Create a head request to get the last modified date from the server
    - Recv the response from the server
    - Parse the response using strtok to get the date and use strptime to get the time back
    - Check the dates using checkModified helper function
    - if it returns 1 meaning it is the most recent version in cache
        - Create a response and send it to the client
        - And send the cache data that is stored at cache ptr found earlier
    - Else
        - If not it will send the request made by the client to the server
        - And recv the response given by the server
        - Send that response to the client
        - It will pull the response type returned by the server and check if it is 200
        - If it is 200
            - It will parse the response from the server and find the length and the last modified date using strtok
            - It will find an empty cache using findEmptyCache from DataStructures.h which will return a pointer to the cacher
            - Then inserts file name time and file length that it pulled from the response into the cache using insertFileToCache from DataStructures.h
            - It finds the index where the response header ends
            - If length given is less or equal to the cache size set in the shared data struct
                - Create the tm data structure and use strptime to take the time given from the response header
                - Initialize i which is a holder of where the cache is going to write at
                - Than iterate through the response of the server till either the cache is filled or it reaches the end of buffer, setting each byte of the buffer to the cache data at index i
                - Increment i every iteration of the loop
                - Calculate if there are any more bytes to be read and set it to bytes
                - If bytes is greater than 0 loop till bytes is 0
                    - Recv another batch of data
                    - And send it to the client

            - Decrease bytes by the amount of bytes read
            - Than iterate through the buffer and set it to cache data
        - If the size of file is too big for the cache
            - Calculate the bytes that need to be read still
            - If so loop till all the bytes have been received and send it to client
            - Decrease bytes needed to be read
    - If the response from the server was not 200 ok increment server fails counter which is an variable for the server pointer

Check_health function

- Checks health for given data structure ServersList server
- Opens connection with port given in ServesList
- If serverfd is -1 set disable to 1 for the server close connection and return
- Else set disable to 0
- Create the health check request and send it to the server
- Recv the and set it to a buffer
- Parse the buffer and get the response number
- If the response number is not 200
    - Set disable to 1
    - Close connection
    - And return
- Parses the body of the response using strtok and set the requests to and fails returned from the body
- Close connection and return