Sehej Sohal sssohal 1650056

Assignment 3 - HTTP-Server with Caching Design Document

1 Goal

The goal of this assignment is to extend the HTTP Server that we have been working with to support a form of caching. The server should be able to cache data from both GET and PUT requests, and function as if it were reading directly from the file. The HTTP Server should also support informative logging. That is, for every request, write the file being requested to the log, followed by any data(if PUT), and an indicator showing whether or not the file was accessed from the cache or from the disk.

2 Assumptions

- The open/read/write/send/recv system calls have some sort of max amount of bits they can read, regardless of buffer size, and that size differs between OS's. Meaning that to have the code work on multiple OS's, I need to keep the buffer size small.
- Usage of common HTTP paradigms, such as pausing and expecting a 100 Continue before uploading a large file.
- Maximum buffer sizes are 16kb, but we're welcome to use smaller buffers.
- C++ strings and complex data structures are available to use.
- Any method of caching discussed in class is acceptable (specifically FIFO).
- Warnings should be minimized, but some are irrelevant (variable length arrays for example).

3 Design

The design of the core HTTP server is exactly the same as from assignment1. The documentation for that assignment is here: https://gitlab.soe.ucsc.edu/gitlab/cse130/fall19-01-group/sssohal/blob/master/assignment1/DESIGN.pdf. Building off of that assignment, some key features have been added:

Logging - Logging has been added to the HTTP Server, as in assignment2. For each request, the filename is written to the log, followed by the content length, followed by a hex dump of the received data. An indicator as well has been added to differentiate whether a file was accessed from the cache for from disk.

To implement logging as such, after each request is received and handled a series of checks are performed on: method, return code, and file location. After which the logging function handles writing these to the user specified log file.

Caching - The method of caching used for this assignment is simply FIFO caching. As such, only 4 files can be stored in the cache at once, and they are popped in FIFO order as more files are accessed. This method was chosen as it was the simplest to implement in the context of my HTTP-Server.

This has been achieved by creating a cache class, along with a page class.

Page.h/cpp - This class has no complex data structures, and it only has two fields: filename and content. When the page is constructed, whether that be from a client, file, or new data, it fills both of these fields.

Page has two methods: fill(string) and writeToDisk(). Fill takes a string and fills overwrites the current content with it. This is used for overwriting data that is already a page. WriteToDisk() takes the current filename and contents, and writes the content to file with that name to disk.

Cache.h/cpp - This class contains a C++ std::vector object, constructed of pages. It also has a max size field, along with a current size field. The vector is handled as a psuedo-queue, and contains the pages that are currently stored in the cache. The decision to use a vector instead of a queue here is that we still need to access all the members in the vector, along with treating it as a queue.

The object has two class methods: add(page) and contains(filename). Add adds a page to the cache. If the cache is full, pop the page off of the front and write that to disk. Then, add the new page to the cache. Contains simply returns T/F depending on if that file is stored in the cache.

4 Pseudocode reused code has been omitted

httpserver.h/cpp

```
void asciiToHex(input, output) {
    for each char in input {
        output += " %02X"
    }
}
```

```
void httpserver::writeHeaderToLog(statusCode, method, contentLength, filename,
wasInCache) {
    if logging isnt enabled {
        return
    }
    if statusCode < 400 //success {
        if method == PUT {
            snprintf(header)
        }
        else {
            snprintf(header)
        }
    } else //failure {
        snprintf(fail_header)
    }
    write(header/fail_header)
    if method == PUT {
        writeFileToLog(filename, contentLength)
    }
}

void httpserver::writeFileToLog(filename, contentLength) { //UNUSED WITH CACHING
ON
    open(file)
    for each 20 chars{
        har* input
        char* output
        asciiToHex(input, output)
        snprintf(format)
        write(format)
    }
    write(line break)
}

void httpserver::writeCachedFileToLog(filename) {
    if !cache.contains(filename){
        return
    }
    content = cache[cache.contains(filename)].getContent()
    for each 20 chars{
        har* input
        char* output
        asciiToHex(input, output)
        snprintf(format)
        write(format)
    }
    write(line break)
}
```

```
void http_server::handle_get(...) {
    ...
    if cachingEnabled == true {
        if file in cache {
            read file from cache
            write(client)
        } else {
            read file from disk
            write(client)
            add file to cache
        }
    } else {
        ...
    }
}

void http_server::handle_put(...) {
    ...
    if cachingEnabled == true {
        if file in cache {
            overwrite file to cache
        } else {
            add file to cache
            overwrite file to cache
        }
    } else {
        ...
    }
}
```

page.h/cpp

```
class page {
    filename
    contents

    page(filename, contentLength) { // fill from disk
        filename = filename
        open(filename)
        fillFromClient(filename, contentLength)
    }

    page(filename, client, contentLength) { // fill from client
        filename = filename
        fillFromClient(client, contentLength)
    }

    writeToDisk() {
        open(filename)
        write(filename, contents)
        close(filename)
    }
}
```

cache.h/cpp

```
class cache {
    vector<page> psuedoQueue
    maxSize
    currentSize

    cache() {
        maxSize = 4
        currentSize = 0
    }

    cache(size) {
        maxSize = size
        currentSize = 0
    }

    add(page) {
        if currentSize == maxSize {
            oldPage = psuedoQueue.popBack()
            oldPage.writeToDisk()
            psuedoQueue.addToFront(page)
        } else {
            psuedoQueue.addToFront(page)
            currentSize++
        }
    }

    contains(filename) {
        if psuedoQueue.contains(filename) {
            return true
        }
        return false
    }

}
```