CSE351: Computer Network
Homework 2

Student Name: Nguyen Minh Duc
Student ID: 20202026

1 Execution Environment

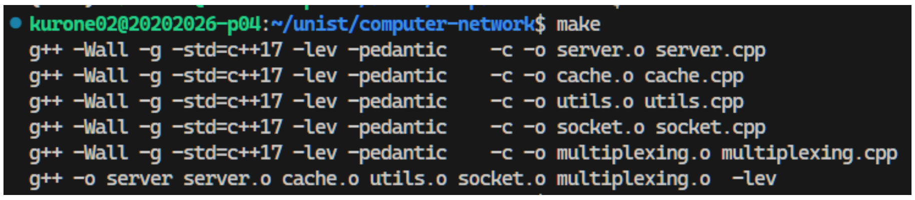
In this homework, my execution environment is the same as the previous homework, i.e., a `ThinkStation` running Windows Subsystem for Linux (WSL2) on Windows 11 Pro for Workstations 64-bit. The `g++` compiler's version is 11.4.0 installed in the Linux kernel. The summary of the system information can be found in Table 1.

Hardware	ThinkStation
Operating system	WSL2 on Windows 11 Pro (Ubuntu 22.04.3 LTS)
g++	11.4.0
Language usage	C++17

Table 1: Execution Environment

2 Compilation Commands

To compile the program, one must install all dependencies. In this homework, we can only use one external library, namely, `libev` or `libevent`. In my case, `libev` is utilized. The following command can be used for installation: `sudo apt install libev-dev`.
All the necessary commands for compilation are included in the file `Makefile`, one can just type `make` in the same directory as the server code to compile.



```
kurone02@20202026-p04:~/unist/computer-network$ make
g++ -Wall -g -std=c++17 -lev -pedantic    -c -o server.o server.cpp
g++ -Wall -g -std=c++17 -lev -pedantic    -c -o cache.o cache.cpp
g++ -Wall -g -std=c++17 -lev -pedantic    -c -o utils.o utils.cpp
g++ -Wall -g -std=c++17 -lev -pedantic    -c -o socket.o socket.cpp
g++ -Wall -g -std=c++17 -lev -pedantic    -c -o multiplexing.o multiplexing.cpp
g++ -o server server.o cache.o utils.o socket.o multiplexing.o  -lev
```

Figure 1: Compilation commands

3 Execution Commands

After compiling, an executable file named `server` will appear. Running the server requires 4 positional command line arguments: the server IP, the port number, the web object directory path, and the maximum cache size in KB. There is another optional parameter: the number of threads with a default value of 4 for multi-threading optimization, one could increase or decrease the argument for better performance. Figure 3 shows an example command for running the server at `0.0.0.0:8080` serving content from `web` directory with 10MB of cache size and 8 threads for multiplexing.

```
./server <IP> <port> <web path> <cache size (KB)> [optional: <threads, default=4>]
```

Figure 2: Command line format of the server

```
./server 0.0.0.0 8080 web 10000 8
```

Figure 3: Example usage of the server

4 Implementation Details

The overall system contains 6 modules, each serving different purposes. Each module contains its corresponding `.cpp` and `.hpp` files.

Module	**Purpose**
`constants`	Contains all global constants
`utils`	Contains utility functions
`cache`	Contains the cache system
`socket`	Contains the initialization phase of the server socket
`multiplexing`	Contains the multiplexing system
`server`	Contains the main function

Table 2: Module's objective

4.1 Constants

This file contains all constants that are accessed globally across the project. In this case, there is only one constant, namely, `BACK_LOG`, which is connection-waiting queue size. Figure 4 contains the information of the constant.

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

const int BACK_LOG = 1024;

#endif
```

Figure 4: Global constants

4.2 Utility Functions (utils)

```
#include "utils.hpp"

// Function to convert an IP address to 32-bit integer
in_addr_t ipStringtoInt(const std::string &ip) {

                    ⋮
}

// Function to read content from a file
std::string readFile(const std::ifstream &file) {

                    ⋮
}

// Function to extract the request URL from an HTTP request
// Assume that httpRequest is a GET request
std::string extractGetRequestURL(const std::string &httpRequest) {

                    ⋮
}
```

Figure 5: Implementation of utility functions

This module contains three utility functions that are useful for implementing other functions in the server. Firstly, ipStringtoInt converts a string of an IP address into a 32-bit number. This can be done by splitting the string by the dot character (.) into 4 numbers, the 4 substrings into numbers, and using the bit-wise operator to concatenate their 8-bit representation into a 32-bit number. Secondly, readFile reads content from the web object file, which can be done by using the C++ standard library. Lastly, extractGetRequestURL extracts the requested web object from an HTTP request, assuming that the request is a GET request. This can easily be done by getting the second substring separated by the whitespace. The detailed implementation can be found in utils.cpp file. Figure 5 shows a summary of the file.

4.3 Cache

```cpp
class Cache {
private:
    struct TData {
        int accessCount;
        std::string content;
    };
    typedef std::unordered_map<std::string, TData>::iterator CacheIterator;

    static Cache* singleton_;
    Cache(): maxCacheSize(0), currentSize(0) { }

    size_t maxCacheSize;
    size_t currentSize;
    std::mutex mutex;
    std::unordered_map<std::string, TData> cache;
public:
    static Cache& instance() { static Cache inst; return inst; }
    void setMaxCacheSize(const size_t &maxCache) { maxCacheSize = maxCache; }
    bool contains(const std::string& key);
    std::string get(const std::string& key);
    void insert(const std::string& key, const std::string& value);
    void remove(const std::string& key);
};
```

Figure 6: The Cache class containing the implementation of the Cache system

This module contains the implementation of the cache system. Details on which the algorithm is used in the cache system can be found in Section 5. Figure 6 shows the skeleton of the Cache object that is used globally in the project. Since there is only one cache instance is used in the project, its implementation follows the Singleton pattern.

4.4 Socket

```cpp
1  #include "socket.hpp"
2  // Function to send an HTTP response
3  void sendResponse(int clientSocket, const std::string& response) {
4      send(clientSocket, response.c_str(), response.size(), 0);
5  }
6  // Function to create a server socket
7  int createServerSocket(in_addr_t server_addr, in_port_t server_port) {
8
                          ⋮
22 }
23 // Function to bind a server socket to a port
24 void bindSocket(int serverSocket, in_port_t server_port, in_addr_t server_addr) {
25
                          ⋮
35 }
```

Figure 7: Implementation of the server socket

This module contains how the server socket is initialized with `createServerSocket` and `bindSocket` functions. Figure 7 shows a summary view of the implementation details of the server socket. The first function is used to create a socket for the server with the option to reuse the port to mitigate the long waiting time after the TCP connection is closed, which could be up to several minutes. The latter function is used to bind the socket to the specified port number by using the function `bind` in the `socket` library of C++. There is another function that, however, is not involved in the initialization phase but the Data transfer phase, which is `sendResponse`. This function is just wrapping over the `send` function in the standard library that supports C++ string.

4.5 Multiplexing

```
#include "multiplexing.hpp"

//  Callback function to send an HTML response
void responseCallback(struct ev_loop* loop, struct ev_io* w, int revents) {

                        ⋮
}

// Callback function to handle client connections
void acceptCallback(struct ev_loop* loop, struct ev_io* watcher, int revents) {

                        ⋮
}

// Function to create io multiplexing with multiple threads
void multiThreading(size_t nThreads, std::string web_dir, int serverSocket) {

                        ⋮
}
```

Figure 8: Implementation of the multiplexing system

This module contains how multiplexing works in the Connection Establishment, and Data Transfer phase in the server. Figure 8 shows summarizes the details. There are three functions: `responseCallback`, `acceptCallback`, `multiThreading`. The first function is responsible for how the data is transferred to the client. If the client's request is not a GET request, the server returns a `400 Bad Request` error HTTP response and disconnects. Then, if the requested web object is not found in the web directory, the server returns a `404 Not Found` error HTTP response. Otherwise, the server will send an HTTP response with the starting header as "`HTTP/1.1 200 OK\r\nContent-Type: text/html...`", followed by the web object content. After that, the server disconnects. The second function, namely `acceptCallback`, is responsible for handling client connections. Here, I use `libev` library for IO multiplexing handling. In this function, a new `libev` watcher for the client socket is initialized, the web object is passed to the watcher, and the IO starts. The last function, namely `multiThreading`, is responsible for creating IO multiplexing with multiple threads optimization. Details of how the optimization works can be found in section 5.

4.6 The Server

This file contains the main function to run the server. Figure 9 contains the detailed implementation of the main function. Firstly, the server parses all command line argument to local variables, and initialize the cache object. Then, the server creates a socket, binds it to the designated port, and listens for incoming connections. Lastly, multi-threading optimization is used to handle the Connection Establishment and Data Transfer phase.

```cpp
#include "cache.hpp"
#include "utils.hpp"
#include "socket.hpp"
#include "constants.hpp"
#include "multiplexing.hpp"

int main(int argc, char *argv[]) {

                  ⋮
    // Initialize
    Cache &cache = Cache::instance();
    in_addr_t server_addr = ipStringtoInt(argv[1]);
    in_port_t server_port = stoi(argv[2]);
    string web_dir = argv[3];
    size_t cache_size = stoul(argv[4]);
    size_t nThreads = 4;
    if(argc >= 6) {
        nThreads = stoul(argv[5]);
    }

    // Set the maximum cache size (input is in bytes)
    cache.setMaxCacheSize(cache_size * 1000);

    // Create a socket
    int serverSocket = createServerSocket(server_addr, server_port);
    // Bind to a port
    bindSocket(serverSocket, server_port, server_addr);
    // Listen for incoming connections
    if (listen(serverSocket, BACK_LOG) < 0) {
        std::cerr << "Error: Listening failed" << std::endl;
        close(serverSocket);
        return -1;
    }

    std::cout << "Server listening on port " << server_port << "..." << std::endl;

    multiThreading(nThreads, web_dir, serverSocket);

    // Close the server socket
    close(serverSocket);

    return 0;
}
```

Figure 9: Implementation of the main function

5 Optimization Details

5.1 Caching

The cache system mainly consists of a hash map that contains the web content and tracks how many times it is accessed, the maximum cache size that defines how large the cache is, and the current cache size that stores the total content size stored in the cache. There are functions to check if a web object is in the cache, get the content of the web object in the cache, insert the web's content into the cache the Least Frequently Used removal policy, and remove the web object from the cache. If the new web object's size is larger than the remaining space, find the object with the least usage and remove it from the cache. Keep removing the entries until the new entry fits the cache. Note that if the object itself is larger than the cache, it will never be put into the cache. Figure 10 contains the detailed implementation of the algorithm.

```cpp
// Function to add a key-value pair to the cache
void Cache::insert(const std::string& key, const std::string& value) {
    if(value.size() > maxCacheSize) {
        return;
    }
    std::lock_guard<std::mutex> lock(mutex);
    // Check if adding the new entry exceeds the maximum cache size
    // If yes, remove based on LFU policy
    while(currentSize + value.size() > maxCacheSize) {
        // Least Frequently Used
        CacheIterator min_it = cache.begin();
        for(CacheIterator it = std::next(cache.begin()); it != cache.end(); it++) {
            if(it->second.accessCount < min_it->second.accessCount) {
                min_it = it;
            }
        }
        // Drop from cache
        currentSize -= min_it->second.content.size();
        cache.erase(min_it);
    }
    // Insert to cache
    cache[key] = {1, value};
    currentSize += value.size();
}
```

Figure 10: Implementation of the Least Frequently Used removal policy

5.2 Multi-threading

```cpp
// Function to create io multiplexing with multiple threads
void multiThreading(size_t nThreads, std::string web_dir, int serverSocket) {
    // Create a vector of threads and libev event loops
    std::vector<std::thread> threads(nThreads);
    std::vector<struct ev_loop*> loops(nThreads);

    // Start the threads
    for (size_t i = 0; i < nThreads; ++i) {
        loops[i] = ev_loop_new(EVFLAG_AUTO);
        threads[i] = std::thread([loops, i, &web_dir, serverSocket]() {
            // Create an I/O watcher for libev
            ev_io io_watcher;
            io_watcher.data = &web_dir;
            ev_io_init(&io_watcher, acceptCallback, serverSocket, EV_READ);
            ev_io_start(loops[i], &io_watcher);

            // Run the libev event loop for this thread
            ev_run(loops[i], 0);
        });
    }

    // Join the threads (waits for all threads to finish)
    for (size_t i = 0; i < nThreads; ++i) {
        threads[i].join();
    }
}
```

Figure 11: Implementation of multi-threading optimization

Another optimization used to boost the performance of the server is multi-threading. Figure 11 shows the detailed implementation of the optimization. Here, I create **`nThreads`** (default is 4) threads and each thread initializes and runs its own `libev` event loop. At the end of the function, all of the threads are joined to finish the execution (although this is not reachable as the server runs until it is interrupted by user's input).