Tecnológico de Monterrey

Campus Santa Fe

Advanced Programming

Final Project

Quetzalcoatl Server

Final Project Documentation

Roberto Alejandro Gutiérrez Guillén	A01019608
Salomón Charabati Michan	A01022425
Alberto Ramos	A01374020

Date: December 09, 2019

Profesor: Gilberto Echeverria

Index

Problem

Description

- A light alternative to the Apache server, we decided to create our own web server on C.
- As the browser is our “client” program. We have created a way to push updates from the servers, either by having our client be constantly listening or using what's called “comets”, which forces a browser to accept data that's pushed by the web server, it doesn't need to explicitly request it.
- We are now able to handle client requests from the server. Another challenge will be to be re-render the website with the requests from the client. For example, if the client clicks a button on the page, the server will execute and render the changes that need to be done.

Topics used

The solution uses the following topics seen in class:

- Dynamic memory:
- Pointers:
- Process creation:
- Inter-process communication:
- Signals:
- Threads:
- Parallel programming:

Solution

- Our program can serve as an HTTP server. It will have to serve multiple clients and always be listening to any request done by a client. Our server will also be able to talk

to a client without the client be explicitly requesting it, by using HTTP comets. We will use the library socket.h.

- The server is able to handle GET and POST requests from the browser and send back multimedia like images to the browser.
- We're hosting the projects we've done in class (Game of Life, Calculate PI and Array). so far in the server and the clients can interact with each of them, obviously, each client can interact with a different one or the same one, each in their own process. For example, a client connected to the server might encrypt messages using the vigenere cypher, another one might be playing blackjack with the server (be careful I've heard he likes to cheat), another one might be doing matrix multiplication and another client could be playing with linked lists.
- The bulk of the task is able to handle communication between the server and the clients (browser) but most of the content is already developed.
- Some of the exact ways we are implementing the server were explained previously in the topics section.

Methods and Formulas

Game of Life:

The game is played on its own, with cells arranged in a matrix. Every iteration of the game, the cells can remain as they are, die or be born, following these rules:

Any live cell with fewer than two live neighbours dies as if caused by underpopulation.

1. Any live cell with two or three live neighbours lives on to the next generation.
2. Any live cell with more than three live neighbours dies, as if by overpopulation.
3. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The program takes two or three arguments from the command line:

1. The number of iterations of the simulation to compute.
2. The name of the file that contains the initial setup of the cells.
3. In the case of the program using threads, it should receive a third argument specifying the number of threads to create.

Server PI:

If the client program is interrupted with SIGINT, it sends a message to the server to indicate that he is about to disconnect, and the server sends the current value of pi, along with the number of iterations computed so far. Then the client program prints the result and finish.

If the server is interrupted with SIGINT, it should send a message to all the clients connected, with the number of iterations computed and the current result so far. Once it sends messages to all clients, it can finish.

Random Array:

- Fills an array of size 30 with random numbers in the range from 1 to 100
- The user input a number in the same range
- Computes the difference between the user-provided number and each of the numbers in the array.

Main project functions

Initialize Server (The setup)

int initServer(int port)

This function sets everything up for the server, it starts by declaring the event manager and the connection. These structs contain all the necessary information for the client and the server to talk to each other. The server opens the communication sockets to talk to the client which in this case is the browser. Afterwards, it initializes the signal handlers for the server and the server itself. Then it binds the server to the connection struct and assigns it a port. The next step is setting for the server by establishing the working directory for HTML files and if it's going to show the index at the beginning also by binding the mongoose connection. Subsequently, with a for loop, it creates all the required threads and checks connections for IO readiness with a poll, this loop runs all the time until the server is

terminated. Last but not least, it frees the memory for the mongoose server and closes the sockets.

Event Handler (The boss)

static void ev_handler(struct mg_connection *nc, int event, void *ev_data)

This function is the main orchestrator, it handles all the events and sends signals to the threads to start working. The main event is “MG_EV_HTTP_REQUEST” which according to the redirect it executes different functions, like pi, array or game of life. In each case it gets information from the HTML form, prepares the work struct and sends the struct for the thread, then the thread which is waiting for an event is activated.

Worker Thread

void *worker_thread(void *param)

This is where magic happens, in this function the threads do the hard work which is then sent through a channel to result thread function. Firstly, it initializes the work struct to receive the arguments from the event handler, then result struct to store the results to send to the result function and lastly, it initializes some program-specific variables as you cannot declare inside a switch case.

This function is called at the beginning when the init server creates the threads, however, the function just waits for a signal to be activated, so it doesn’t waste resources. When the thread is activated by a signal from the event manager it reads the information and copies it in the work request struct. Afterwards, it copies from the work request struct into the result struct some information that it remains the same as connection id, type of program and the input number. The next part of the function is the switch that with the type it knows which program to execute. Inside each case, the program-specific function is calculated and the result is assigned. Lastly, after the switch case, it sends a broadcast signal so the result thread function is activated and receives the result struct.

Result Thread

static void result_thread(struct mg_connection *nc, int ev, void *ev_data)

The last part of the puzzle, this is where the result is pushed to the client. This function is inactive until is triggered by the work thread. Firstly it declares the connection pointer and the buffer. Then it cycles through all the connections until it finds the required one. When the connection is the required one the work struct is copied. Afterwards, with the same logic as the work thread, there are switch cases which manages different cases for each program. They work similarly, as they fill the buffer then push the HTML to the client.

Deployment

Requirements

- OpenMP installed
- Gcc compiler

Run the code

1. Download the zip from GitHub
2. Unzip files in any directory
3. Open the terminal and run the following command

gcc multithreaded.c mongoose.c string_functions.c game_of_life_omp.c pgm_image.c -D MG_ENABLE_HTTP_STREAMING_MULTIPART -D MG_ENABLE_THREADS -o multi

4. Run

   ./multi

5. The server should indicate that is currently running by printing “Started on port 8080”

```
Alejandros-MacBook-Pro:FinalProject axguti$ gcc multithreaded.c mongoose.c strin
g_functions.c game_of_life_omp.c pgm_image.c -D MG_ENABLE_HTTP_STREAMING_MULTIPA
RT -D MG_ENABLE_THREADS -o multi
Alejandros-MacBook-Pro:FinalProject axguti$ ./multi
Started on port 8080
```

6. Open any browser window and type the following link

http://localhost:8080/

7. The browser should display the following menu

Index of /

Name	Modified	Size
post-pi.html	08-Dec-2019 18:38	537
post-array.html	08-Dec-2019 18:38	308
preOmp.html	09-Dec-2019 08:41	301
omp.html	09-Dec-2019 08:44	289
differences.txt	09-Dec-2019 08:01	113

Mongoose/6.16

8. Select any of the following HTML's to run a program

- post-pi.html
- pre-omp.html
- post-array.html

References:

https://github.com/cesanta/mongoose

https://www.openmp.org/