16 December 2021
ELFATIHI Maxence
OUASFI Amine

Operating System Projet

Concurrency webserver

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Contents

1 Introduction

The project aim is to develop a concurrent (multithreaded) web server using the provided C code of a non concurrent web server. The initial implementation uses a single thread to handle a request at a time. As a result, a client trying to access a file provided by the webserver can do so only when the server is not busy handling another request.

The concurrency functionality can be implemented through the use of a buffer to store the received requests and a dedicated pool of threads handling the http requests available on the buffer.

2 Non concurrent web server

2.1 Web server architecture

The initial implementation uses various files for the client and the server part which are:

`wserver.c`, `wclient.c`, `io_helper.c` and `request.c`.

The files are compiled using the gcc which is available on a Ubuntu 20.04 LTS machine to generate executables for the server and the client part. The executables can be launched in order to run the server in a single threaded configuration.

The `Makefile` purpose is to compile the required files using `gcc` compiler in order to create the executables for the **webserver**, the **webclient** and the **spin** dynamic content.

This is done using the command:

```
make <name_of_executable>
```

To compile all the executables we use the `make all` or `make` command. To remove the executables we use the `make clean` command.

2.2 The server

The server is implemented using these files: `wserver.c` , `io_helper.c` and `request.c`. We use the following command to run the server:

```
./wserver [-d <basedir>] [-p <portnum>]
```

We specify the working directory after `-d` and the port number that the server will be listening in after `-p`.

2.3 The client

The client is implemented using these files: `wclient.c`, `io_helper.c`. The purpose of the client executable is to send requests to the server. We use the following command to run a client request for the server:

```
./wclient localhost [portnumber] [/filename?variable]
```

A dynamic content file that can spin for **N** seconds can be generated using `spin.c` and will be useful for the implementation part of the project.

The initial implementation does not take into account any security considerations for accessing the files outside the working directory. This issue will be addressed in the implementation of the concurrency web server.

3 Code dependency diagram

3.1 The server

After being launched, the server parses the arguments we provide to it in order to get the port number to use and the working directory and checks whether the provided directory is valid, it starts listening on the provided port. In case of an invalid port or directory the program interrupts and displays an error. The server uses a infinite while loop in order to handle every incoming request. After receiving a request, it handles it by using its file descriptor and closes the connection.

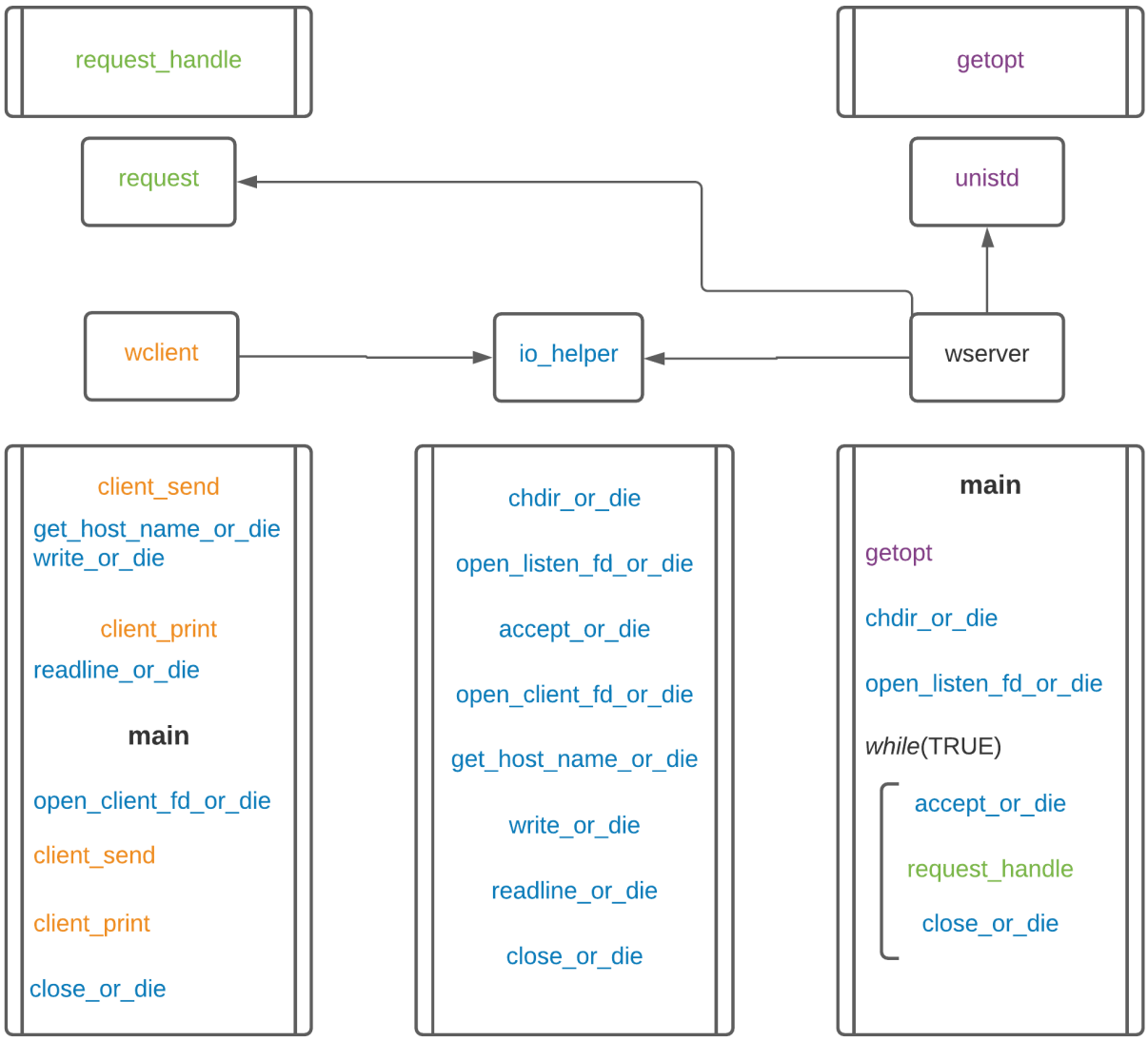

Figure 1: Overview of the different components of the provided code. Colors indicate the file where each function is implemented, arrows show dependencies between files and the block under each filename summarises its structure.

3.2 The Client

After being launched, the client uses the provided arguments which are: the hostname, the port and the filename and opens a single connection to the host on the provided port. After that, it creates and sends an http request for the specified file and displays the http header and content of the response.

4 Multi-threaded server

The main idea of this implementation is to provide a buffer where client requests are stored and a pool of threads that handle these requests concurrently. However, in order to handle the requests the threads need to read from the buffer and to change its state. These operations must therefore be protected so that no other thread has access to the buffer during their execution. We have chosen to implement this protection using `locks`. Moreover, adding a request to the buffer must also be protected in same way. The diagram displayed in figure 2 summaries the structure and dependencies of the our web server.

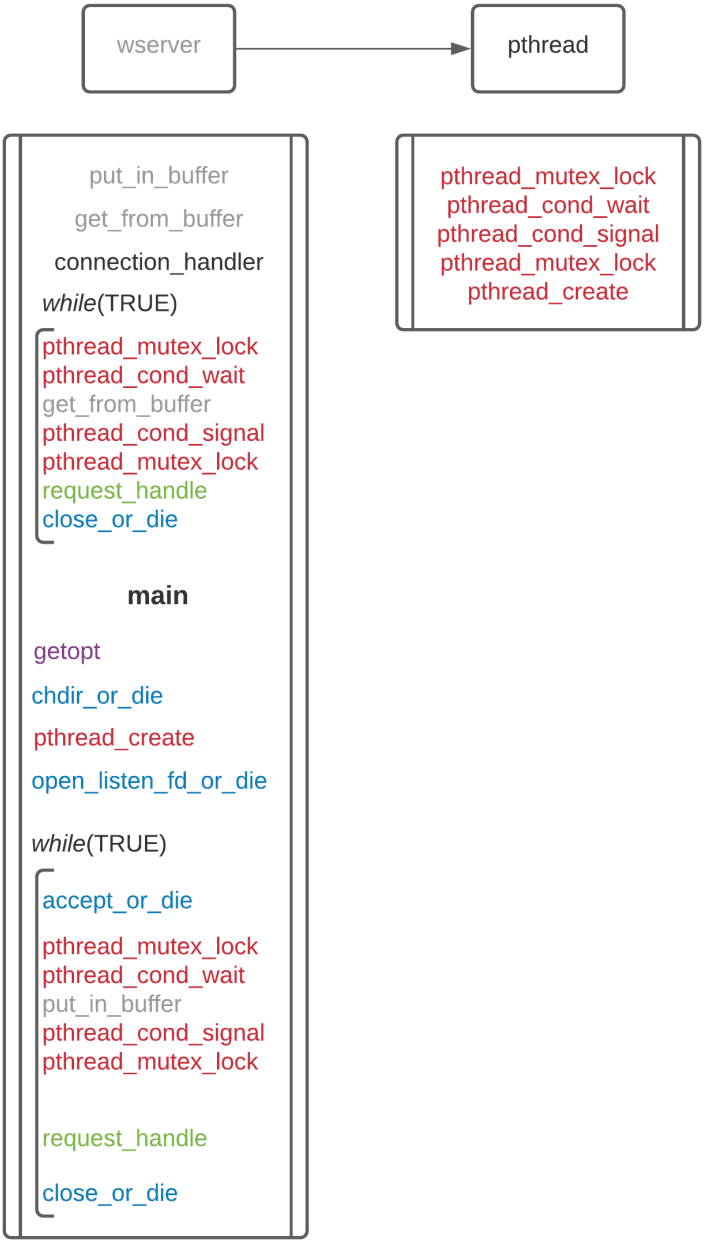

Figure 2: Multi-threaded server dependencies and structure.

4.1 The producer

Our implementation follows the consumer/producer model. A `master thread`, implemented in the `main` function, receives requests and calls the function `put_in_buffer` to add them to the buffer. The execution of this call is protected by a `lock`. When the buffer is full, the lock is atomically released and the `master thread` is

Algorithm 1 Main function and producer

```
initialize server arguments: buffersize, root_dir, port, n_threads, count
initialize mutex lock and condition variables empty and fill
update server arguments based on argv
initialize threads
for thread in threads do
    pthread start thread
end for
start listening to port port
//here starts the producer
while True do
    Wait for an incoming request and store it's file_descriptor when accepted.
    Enable mutex lock
    while buffer is full do
        Release lock and block the current thread on the condition variable empty;
    end while
    put request in buffer and increment count
    unblock at least one of the threads that are blocked on the condition variable fill
    disable mutex lock
end while
```

blocked on the condition variable `empty`. Otherwise, an incoming request is added to the buffer and a signal is sent to the consumer threads waiting on the condition variable `fill`.

4.2 The consumer

Algorithm 2 Consumer

```
while True do
    Enable mutex lock
    while buffer is empty do
        Release lock and block the current thread on the condition variable fill;
    end while
    get request from buffer and decrement count
    unblock at least one of the threads that are blocked on the condition variable empty
    disable mutex lock
    handle request
    close or die
end while
```

The `empty` signal is implemented on the consumer side in the function pointer `connection_handler`. This function is used to create the consumer threads that will handle the requests previously stored in the buffer. First, a request is fetched from the buffer with the function `get_from_buffer`. The returned **file_descriptor** is then used to handle the request and close the connection. Only the first operation (`get_from_buffer`) is protected by a `lock`. In the protected block, the `lock` is atomically released if the buffer is empty and the calling thread is blocked on the condition variable `fill`. Once a request is fetched from the buffer an `empty` signal is sent to the master thread.

Concerning the buffer, we initialize it using `malloc` function so as to define it as a global variable which size is defined by the user. In addition, our we implement it as queue following a FIFO (First In First Out) approach.

5 Tests and results

5.1 Non concurrent web server

To test our implementation we first ran the non-concurrent web server with one and 10 clients and measured its execution time with a request on `spin.cgi` file for **5 seconds**.

For the single client request we used the following command:

```
time /wclient localhost 10000 /spin.cgi?5
```

Furthermore, to run multiple client requests simultaneously we decided to use `xargs -P <N>` bash command to launch **N** client processes at a same time instead of multi-threading the client requests.

```
time  seq  <N> | xargs -n 1 -P  <N>  -I{}  ./wclient localhost 10000 /spin.cgi?5
```

As expected, the first command got executed in 5.171s, whereas the second one took approximately 52.648, seconds.This means that client requests are executed one after the other. Hence, the execution time grows linearly with the number of client requests.

5.2 Multi-threaded server

We tested our multi-threaded web-server on different settings in order to see how its execution time evolves when the buffer size, the number of clients or the number of threads changes.

Table 1: **Multi-threaded web-server execution time.**

Test	buffer size	n_clients	n_threads	execution time
A	8	9	10	5,446s
B	8	9	100	5,126s
C	8	90	10	45,464s
D	8	90	100	6,641s
E	80	9	10	5,114s
F	80	9	100	5,111s
G	80	90	10	45,218s
H	80	90	100	5,975s

Our results are displayed in the table 1. They show the execution time of each test, averaged over 5 executions. Firstly, it is noteworthy that when the number of threads running is greater than the number of clients (tests A, B, D, E, F, H), the execution time is very close to that of a single client request. Moreover, when the number of clients exceeds the buffer size (tests D, H), we can see a slight increase in execution time. This means that the server processes several requests at once, but suffers from a slight slowdown when the buffer size is very small. Conversely, running the server with fewer threads than incoming requests incurs a very high execution time that grows linearly with $\frac{n_{clients}}{n_{threads}}$ instead of $n_{clients}$ as for the single thread server.

The server is launched with following command:

```
./wserver [-d <basedir>] [-p <portnum>] [-b <buffersize>]  [-t <n_threads>]
```

In addition, we used the same command we presented previously to run many clients simultaneously.

6 Conclusion

We learned a lot during this project. First about multithreading and how to implement it using the `pthread` library of the C programming language. Then we have also seen how it is possible to protect reads and writes in the buffer in a multithreading context. We chose to do this using locks but we could also have used semaphores. The main technical difficulty in this project concerned the placement of the locks. That is to say defining which instructions to protect in read/write mode. Moreover, we had to set up tests where the server receives several requests at the same time and automate this with bash. From an organizational point of view, the management of time and tasks was not very straightforward as the time allocated to the project was very short.