OS Multi Threads Project Report

Group members

KOUAKOU Kouadio Aristide

SILUE Ouawortiè Salimata

Teachers

Guillaume DOYEN

Renzo NAVAS

Date : 1 décembre 2021

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

TABLE OF CONTENTS

CONTEXT OF PROJECT

This document presents the Operating System project related to threads and the aim was to apply the studied concepts seen in class. This project is realized in pairs.

PROJECT MAIN OBJECTIVES

The objectives of this project were to :

- To learn the basic architecture of a simple web server
- To learn how to add concurrency to a non-concurrent system
- To learn how to read and modify an existing code base effectively

To reach these, we added one key piece of functionality to a basic existing web server which works in mono-thread. In other words, we made a web server multi-threaded in order to handle multi requests from different clients.

EXISTING

We had an existing code base which is a simple mono-thread web server which structure is the follow :

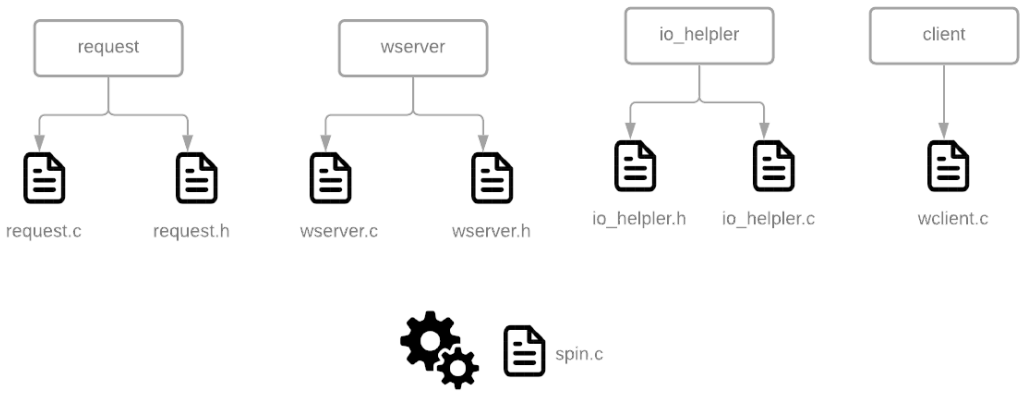

Figure 1 : Global structure of existing code

- **wserver.c**: contains main() for the web server and the basic serving loop.
- **request.c**: performs most of the work for handling requests in the basic web server. Start at request_handle() and work through the logic from there.
- **io_helper.h** and **io_helper.c**: Contains wrapper functions for the system calls invoked by the basic web server and client.
- **wclient.c**: contains main() and the support routines for the very simple web client.

- **Makefile**: We also provide you with a sample Makefile that creates wserver, wclient, and spin.cgi.

The existing one, contains a file called **spin.c** which is a simple CGI program callable by the client during a request.

FUNCTIONING

Functioning : Server

Our server, as we said, is a **multi-threads server**. It creates by a master thread a pool of threads which wait for requests. The master thread is responsible for accepting new HTTP connections over the network and placing the descriptor for this connection into a fixed-size buffer; the master thread doesn't read from this connection.
A worker thread wakes when there is an HTTP request in the queue; when there are multiple HTTP requests available, which request is handled depends upon the scheduling policy, which is in our case FIFO.
The number of threads, the buffer size and the scheduling method are given parameters of the server. The following figure presents the main structure (organigramme) of this server.

The diagram on the right, shows how the server works when it starts up.

1 - When the server starts, it looks for parameters. If the entry parameters are incorrects (Ex: if a port is is a) the server return with a error otherwise it go **2**.

2 - If the parameters are correct the server call the master thread

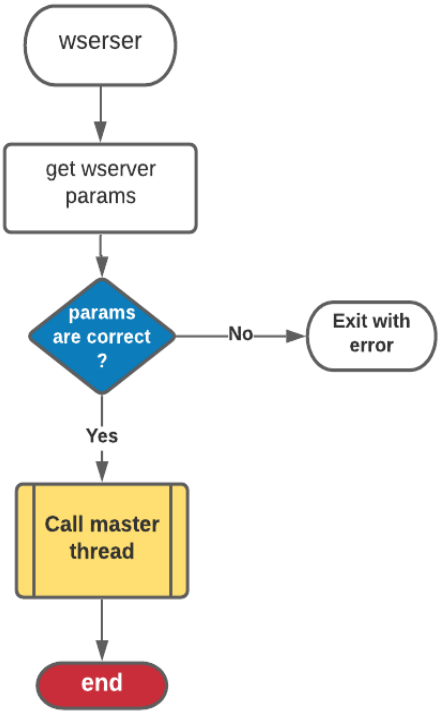

Figure 2: server global diagram

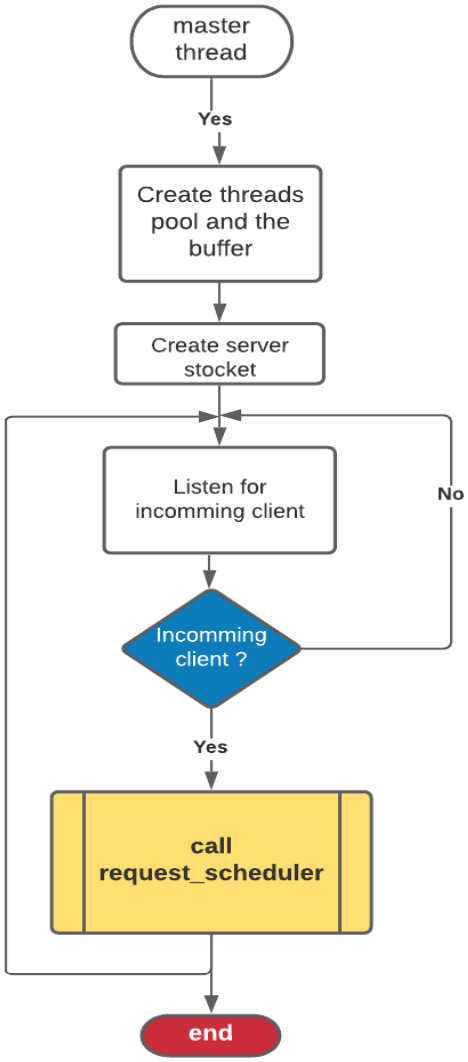

Figure 3: master thread functioning diagram

The diagram at the left resumes the master worker functioning . It creates the pool of threads and the server socket. Afterward, it listens for incoming clients. When we have a client, the master thread calls the scheduler. To insert it in the request buffer.

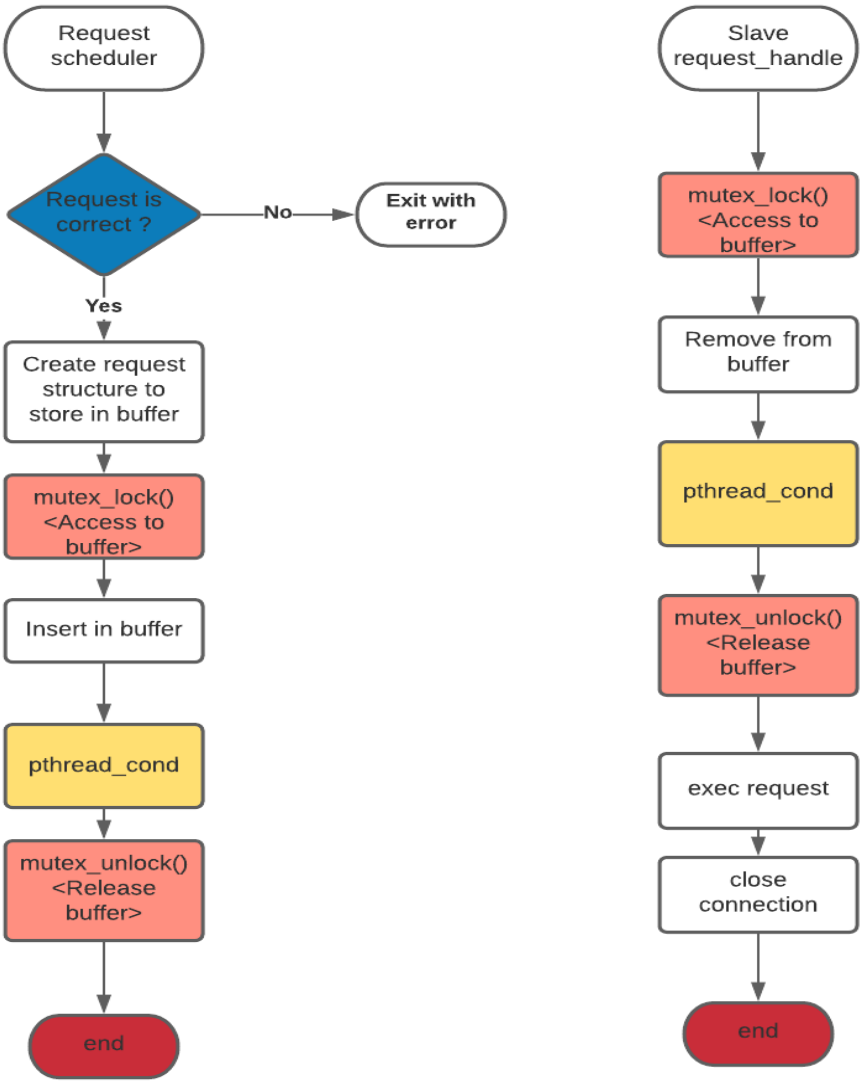

Figure 4: client functioning diagram

The diagrams above show respectively request_scheduler and slave_request_handle functions. The request scheduler inserts in a buffer with the FIFO policy. We use mutex and cond to deal with buffer access because it is a shared variable between threads. Concerning the request_handle the goal is to serve either static or dynamic requests.

Functioning : Client

The client is now a multi-client. The idea is two test how our multi server works so we need to simulate a multiple client scenario. This handles simultaneous requests. It is called **multiple_clients.bash**

```
echo Started

        for i in {1..10}
        do
                ./wclient localhost 30000 /spin.cgi?$i &
        done

echo Finished
```

PERFORM TESTS

The tests were on both dynamic and static requests. And the server responds well to these requests. During the test we run the bash client script which as we mentioned on top, creates simultaneous clients. (The results could be see in demo)

CONCLUSION : LEARNING AND DIFFICULTIES

Learning :

The project was itself more profitable for us. We learnt in fact :

- How to deal with multi-thread
- Concurrency around variables
- Structure and table in C
- How to work with code structuration
- And how to debug code

Difficulty:

The main difficulties were in :

- segmentation fault with C.
- Logic problem some time (with how to structure the scheduler)
- And the last transmit arguments to threads