Part 2: Multi-Threaded Web Server

Introduction

In this part of the project you will:

1. Understand the basics about threading and synchronization.
2. Get familiar with the use of external C libraries.
3. Design and make modifications to the previously implemented server.
4. Test your implementation with some test cases.

Deadlines and Evaluation

The project needs to be implemented and tested by **Deadline 3 (FRIDAY SEPTEMBER 30th, 23:59pm)** under the **Assignments** section on StudentPortalen.

Getting Started

Background Material

Before you get started we strongly recommend you recap the material given during the parallel programming lectures.

Kickstart

From the course website you will be able to download the tarball **part2-kickstart.tar.gz** which contains the following files:

- `README`: *Contains a small description and the project license.*
- `Makefile`: *Main makefile used to compile the project with GNU Make.*
- `utils.c, utils.h`: *A module containing functions used mainly for I/O, pretty printing and and interaction with the operating system.*
- `request.c, request.h`: *Module used to process the requests from the clients.*
- `server-mt.c`: *File where the server is implemented.*

Step 0: Tutorial (Individual)

Before moving to the next step, make sure to have followed the tutorial/exercises from the parallel programming lectures.

Step 1: Designing threading changes

Now that you are familiar with the basics of threading and synchronization, we will try to improve our previous implementation to make use of multicore architectures and improve performance.

Motivation

The server implemented in **Part 1** behaved as we expected: a client sends a requests, the server receives it, process it and send a response back. To attend multiple clients, a simple queue is held where the requests are enqueued, as shown in **Figure 1**. However, we have seen recently the introduction of multicore architectures in the market. This means that nowadays, processors have several cores capable of making computations at the same time.

Let's say that we are running our server on a quad-core processor machine. The server will be launched and executed in one of the cores, while the others are free to do other tasks at the same time, or be idle in the worst case.

Now, suppose a new client connects and sends a request for a file that is 2GB. It will take a considerable time until the transfer is over. During this time, other clients might try to send requests to the server, but unfortunately, since the server is busy transferring a fat 2GB file, it won't be able to handle new requests, even if any of the other cores in the processor is idle!

A better approach would be to have a server that is capable of delegating the work somehow, to smaller work units, each of them in charge of requests from different clients. Then, they get executed when hardware units are free, no matter if we have only one core, or several of them.

Figure 2 shows this in simpler terms. Imagine the server as a shop, where there is a first an information desk with an agent. As clients come into the shop, the agent gives a number to the clients, queuing them somehow and sending them to wait in a waiting area. Several counters are available to attend the clients, and whenever they are free they call a new one in.

By taking this approach, we can now have shorter waiting times when multiple requests arrive at the same time, as we are able to process the in parallel.

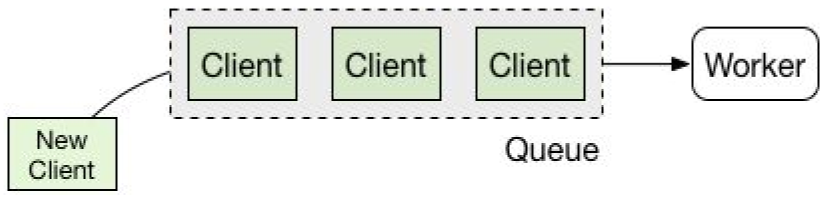

Figure 1. Single Threaded Server.

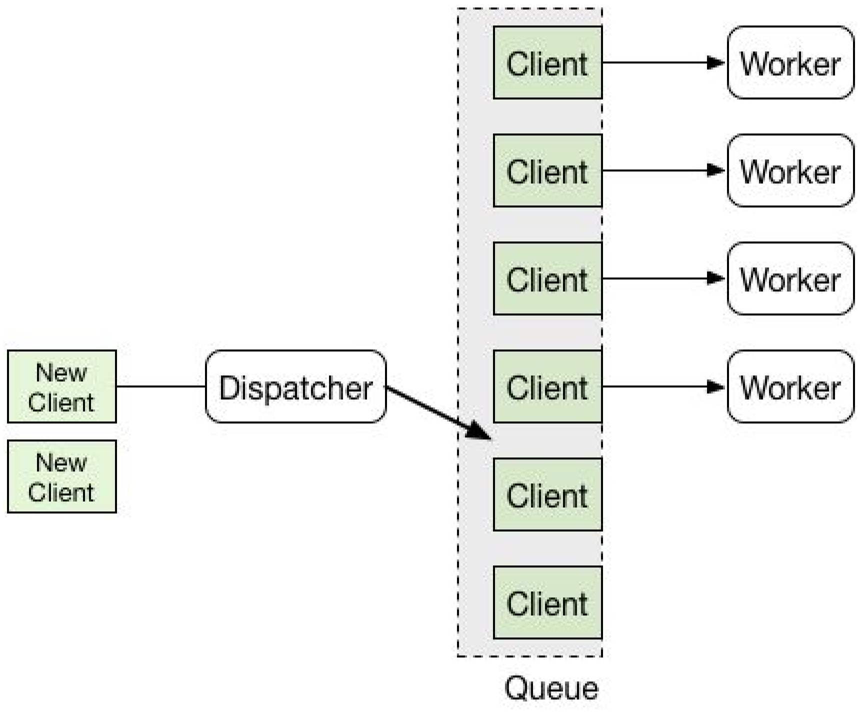

Figure 2. Multi-threaded Server.

Scheduling policy

The fashion in which the clients are ‘removed’ from the queue or waiting list can make an impact in some server metrics such as average response time, average queuing time, total waiting time, etc. This is usually called the *scheduling policy*, and the performance of the server is somehow tied to it.

In this project, you will be given a particular scheduling policy to implement, and we will compare the results later to analyze both their advantages and disadvantages.

The scheduling policies we will consider are:

- FIFO (First in-First Out): This is probably the most intuitive one, as we encounter it in many aspects in our lives. Here, clients are attended according to their arrival order (a regular queue).

- STACK (Last in-First Out): Similar to the previous one, but serving the last client that arrived first.
- SFF (Smallest File First): In this policy, clients are attended according to the size of the file they requested. The client requesting the smallest file will be treated first.
- BFF (Biggest File First): Analogous to the previous one, but serving the biggest file first.

Step 2: Implementation

You will start by making some modifications to the code you previously implemented in Part 1.

In the new request.h, you will see the declaration of a new **thread** structure. It will represent internally each of the threads spawned by the server, that will handle the clients. The structure will hold a **thread id**, together with some counters for some statistics.

This structure will be filled in when spawning a new thread, and modified by each of the requests handler functions when executing a request. To be able to do that, the request functions need access to this structure. Hence, it is necessary to provide it as an extra parameter when calling them.

Your first job is to include this structure as a one of the arguments of several functions. You will see `/*TODO*/` markers to identify where to modify the code. The functions where you need to propagate this argument are:

```
requestHandle()
requestSeverStatic()
requestServeDynamic()
```

With these modifications, the modules can now handle requests knowing details about the thread processing it. You will note that now, the functions `requestServeStatic()` and `requestServeDynamic()` print some statistics about the requests being processed.

Next, you will need to do several modifications to the main server loop to support multiple threads. As you have seen previously, it is necessary to have an entry point function when creating threads.

The strategy will be to have the main loop of the single-threaded server as the entry point function in the multi-threaded version. This is why you will see the declaration of a **consumer** function in `server-mt.c` (line 107).

To code this function, you need to assume that it will be executed right after the thread is created. You will see several **TODO** markers.

The first two things you will need to do is to create a **thread** structure that we declared on request.h, and initialize it. This will identify the thread running and gather some statistics when executing.

Right after that, the main loop of a thread starts as follows:

1. The thread checks if there is any client waiting to be served. If there is not, it waits until there is one with *pthreads* wait primitives.
2. It gets the request from the right client according to the scheduling policy.
3. The request is dispatched to the **Request** module.
4. The connection to the client is closed.

Note that all the previous steps are properly synchronized acquiring and releasing a lock to the shared structure where the incoming requests are stored (you need to add these synchronization too).

Once you reached this point, the entry point function is now done, and this will determine what will be executed by the worker threads (i.e. the ones handling the requests).

To wrap up, there is one final step to accomplish, that is the creation and administration of the threads by the main server loop.

If you look now at the `main()` function in `server-mt.c` (line 194), you will see that several new variables are declared this time:

- `threads`: number of worker threads that the server will spawn to handle requests in parallel.
- `buffers`: number of clients that can be queued (size of the queue).
- `alg`: the scheduling algorithm

Note that in this case, the server allows to have a different number of working threads than clients queued and waiting to be served (buffers). Another alternative would have been to select the same number of threads as clients queued, only accepting new clients when a thread is idle. This makes sense in scheduling algorithms like FIFO, but as we will see later, in scenarios like SFF, having more clients queued can make a difference in the server statistics.

You will also notice that there are global variables declared:

- `max`: this variable will store the size of the queue.
- `fillptr`: the index to the next empty slot in the queue
- `useptr`: the index to the next request to be handled in the queue.
- `numfull`: number of clients waiting to be served.
- `algorithm`: the scheduling algorithm selected by the user

The main server loop first creates a queue (request buffer) with enough room for the number of clients given by the user (the max variable). It will then create the number of *worker threads* specified by the **threads** variable. Once spawned, these threads will start executing the **consumer** entry point function that you fixed previously.

On the other hand, the main server loop will keep running, receiving and queuing the clients appropriately. After accepting the connection of a new client, you will have to implement the rest of the loop as follows:

1. The thread first checks if the request queue is full. If it is, it waits until there is at least one empty slot.
2. Once a new slot is free, a new request structure is allocated and filled with the client's details.
3. The request is added to the queue according to the scheduling policy.
4. The global counter of queued clients is increased.

Note that these steps are also properly synchronized to avoid data races on the shared variables.

Questions:

1. Why is the connection to the client not closed in this case?
2. What are the consequences of adding a call to `join()`, or a barrier after each loop iteration in the main loop?

Congratulations! You have now implemented a parallel version of the web server you had before. Well done!

Step 3: Testing

To verify the server is running properly, repeat the tests on Step 4 of Part 1. The modifications you have done are just internal, so the behavior of the server towards the tests should not change.

Revisions

Version 1.0 – Fall 2015 – Germán Ceballos
Version 1.1 – Fall 2016 – Germán Ceballos