Operative systems project

Andrés Ricardo CÉSPEDES GARCÍA, Qichao HU

December 2021

1 Existing code and objectives of the Project

In the operative systems class we were given a non-current web server. which suffers from a fundamental performance problem: Only a single HTTP request can be serviced at a time. Therefore, in the case where we have more clients trying to access the web server, they will be forced to wait until the current http request has finished. For this project, we were asked to correct this problem with the implementation of multithreading for the server.

1.1 Multithreaded web server

In order to achieve the multithreaded implementation, the next instructions were given: First we should start with a master thread that creates a pool of worker threads. The master thread is then responsible for accepting new HTTP connections over the network and placing the descriptor for this connection into a buffer; Each worker thread is able to handle both static and dynamic requests. A worker thread wakes when there is an HTTP request in the queue, taking into account that the master and the worker threads are in a producer-consumer relationship and require that their accesses to the shared buffer be synchronized.

1.2 Security Considerations

It was asked to take into account the possibility of malicious intentions on our server. The server should reject the possibility of accessing through the file system hierarchy, so we were asked to constrain any pathname with ..

2 Solution

2.1 Flow Chart

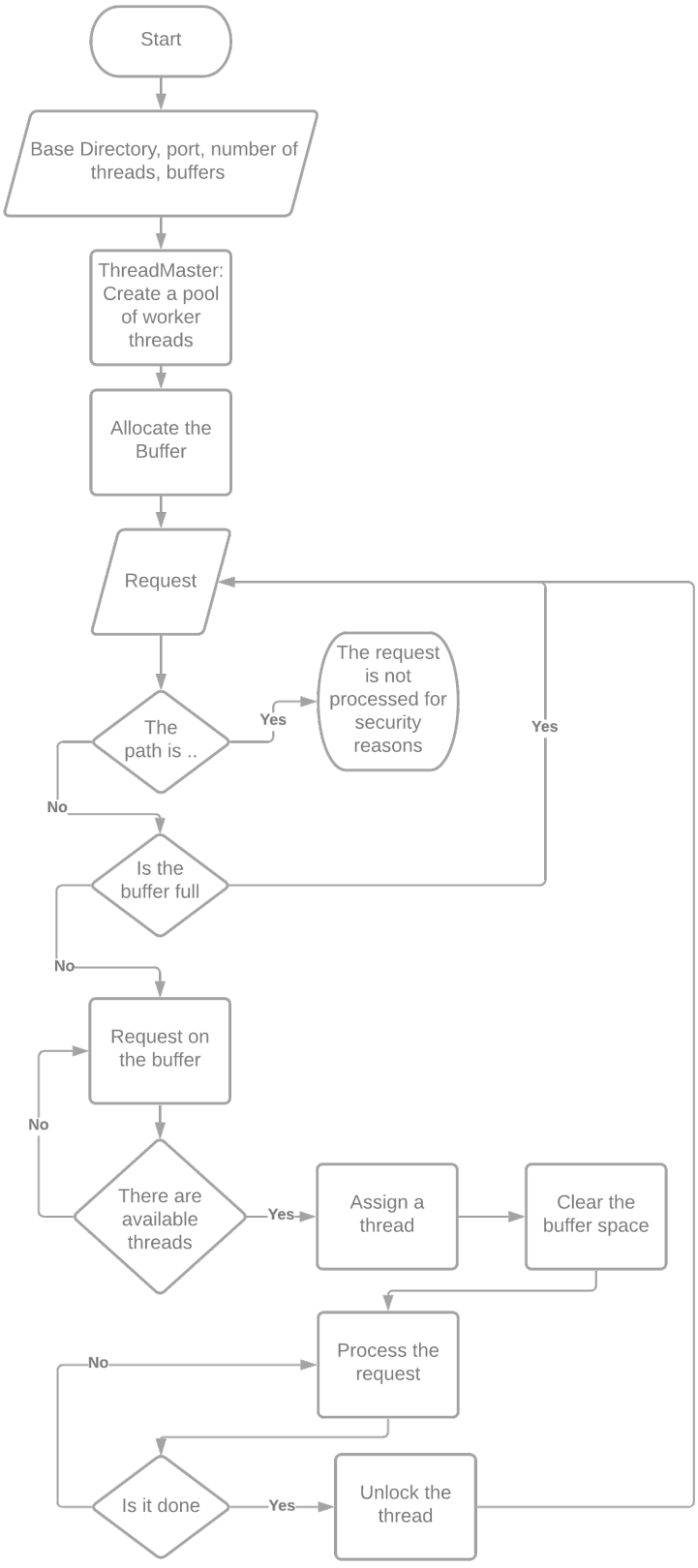

Figure 1: Flow chart of the multithreaded webserver

2.2 Explanation

The project is a multithreaded web server which is capable of handling multiple requests at the same time. In order to achieve this functionality the workflow presented on the Figure 1 was made, and it is furtherly detailed as it follows:

The web server is invoked with some input values which are:

```
prompt> ./wserver [-d basedir] [-p port] [-t threads] [-b buffers]
```

Where the **d** flag corresponds to the root directory from which the web server should operate, **p** is the port number that the web server should listen on, **t** is the number of worker threads that should be created within the web server and **b** corresponds to the number of request connections that can be accepted at one time.

With the values introduced by the user, we create a we created a master thread who is responsible for creating a pool of working threads. The amount will be the same given by the user. The master thread is then responsible for accepting new HTTP connections over the network and placing the corresponding description for this connection into a buffer. We also created a single fixed-size buffer (that we will use a circular one) with a size specified by the user. When we have a request on the buffer queue, a quick security check is made: we made sure of constraining the file requests to stay within the sub-tree of the file system hierarchy, rooted at the base working directory that the server starts in, so if there is any pathname which includes .. the server lets the client know that his request is not going to be processed.

After the security considerations, we check the status of the buffer. If it is full, this means that we are not able to handle any more requests until the buffer is cleared. If we have enough space to receive a request on the buffer, we will add the request on the buffers queue. Then we check if there is a working thread available to handle this request, if so, we clear the space that this request was occupying in the buffer while a worker thread from the previously created pool wakes and starts its work.

The worker thread is able to handle both static and dynamic requests, and its job is to perform the read on the network descriptor, obtaining the specified content of the request and then return the content to the client by writing to the descriptor. Once this process is done (the request has been handled), the thread is unlocked and allocated back to the pool, where now it is ready and waits for another HTTP request.

We consider important to mention that the consumer thread and the master thread are on a consumer-producer relationship and both of them access to the same buffer in a synchronized way.

3 Essential algorithms

Algorithm 1 Master_Thread

Input: A struct of the command input information **args*
```
1: begin
2: for i ← 0 to the number of threads do
3:     Create one Worker_Thread;
4: end for
5: Make a listening socket;
6: while True do
7:     Wait for a request;
8:     Lock the thread;
9:     Store the request in the buffer;
10:    while the buffer is full do
11:        Retry to store the request in the buffer;
12:    end while
13:    Wake up a Worker_Thread;
14:    Unlock the thread;
15: end while
16: end
```

Algorithm 2 Worker_Thread

Input: A struct of the command input information **args*
```
1: begin
2: while True do
3:     Lock the thread;
4:     Take a request from the buffer;
5:     while the buffer is vide do
6:         Sleep; // Wait for the signal from Master_Thread
7:         Take a request from the buffer;
8:     end while
9:     Unlock the thread;
10:    Handle the request;
11:    Close the request;
12: end while
13: end
```

Algorithm 3 fill_buffer

Input: Request
Output: The status of buffer(1 means buffer is full) **begin**
```
1: if actual_buffer_size = buffersmax then
2:     return 1;
3: end if
4: // The tail values reset to 0 when the maximum size is reached.
5: tail ← (tail + 1) mod buffersmax;
6: // Put the request in the buffer.
7: buffer[ tail ] ← request;
8: actual_buffer_size++;
9: return 0;
10: end
```

Algorithm 4 clear_buffer

Output: Request
```
1: begin
2: if actual_buffer_size = 0 then
3:     return NULL;
4: end if
5: // Take the request from the buffer.
6: request ← buffer[ head ];
7: // The head values reset to 0 when the maximum size is reached.
8: head ← (head + 1) mod buffersmax;
9: actual_buffer_size--;
10: return request;
11: end
```

3.1 Explanation of the algorithms used

As we already mentioned, everything starts with the input of the user, when he gives the program the path where it wants to work, the port that he user wants to use, the number of threads that will be used and the size of the buffer that will be used in order to accept multiple connections at the same time.

Then, the server starts by creating a pool with the master thread, which corresponds to the **Algorithm 1**. In this algorithm, we receive the information provided by the user, then we do a little cycle where we are going to create the amount of threads that was received from the user. After that, we use the port that the user introduced for the web server.
The master thread is then responsible for accepting new HTTP connections over the network and placing the descriptor for this connection into a fixed-size buffer (with the size specified by the user), to do that we wait for a request and as soon as we receive it, we lock the thread, if we have enough space, we store it in the buffer. After being stored in the buffer, we wake up a worker thread that will handle the request and as soon as it is finished, the thread will be unlocked.

Now we have to address the **Algorithm 2** that corresponds to the worker thread. The worker thread is called from the master thread and it starts by locking a thread (so that we don't interrupt the process), then, if we have any requests stored in the buffer, we unlock the thread that was being used, we take the request from the buffer and clear the space that this particular request was occupying by handling the request. In the case that we dont have any particular request on the buffer, we wait for one.

Notice that the master and worker threads are working in a producer-consumer relationship. As we discussed in class, this is a standard issue in concurrent programming. In order to assure its correct working, we have to use a single shared buffer that will be accessed in the order given by a scheduling algorithm.
In this particular case, we decided to implement a First In, First Out algorithm for simplicity reasons. In order to implement it, a circular buffer was recommended to assure that the program will run smoothly [1].

After adapting the code that was explained in [1], we required two functions that were responsible of filling and clearing the buffer in a circular. The filling function is the one represented in the **Algorithm 3**. In this algorithm we first check if the buffer is full, if we have space, we move the "tail" of the circular buffer, we write the information on the buffer and increment the counter that we use in order to know how many elements are stored on the buffer.

A similar process is carried out with the **Algorithm 4** in charge of clearing the buffer: First we check if there is something to clear from the buffer, if that is so we proceed to delete the oldest data storaged in the buffer and then we reduce the counter that we are using in order to know the actual amount of elements stored on the buffer.

4 State of the code

At the end of the project we have a multi-threaded web server which is capable of handling static and dynamic requests. The amount of threads, the buffer, the base directory and the port are selected by the user.

4.1 Tests

4.1.1 normal_test.sh

This bash file will test the normal process. First, open a new terminal and start the server which has 5 threads and 5 buffers.Then open another terminal with 4 tabs, each tab terminal sends a dynamic request. There are 4 dynamic requests in total. At the end, open another terminal with 5 tabs, each tab terminal sends a static request. There are 5 static requests. If the server works well, there will be 4 threads to handle dynamic requests and the last thread handles all 5 dynamic requests.

4.1.2 occupy_all_threads_test.sh

This bash file will test the situation that all the threads are occupied by dynamic requests to see if the next requests can be safely stored in the buffer and be handled after the dynamic requests are done. First, open a new terminal and start the server which has 3 threads and 3 buffers. Then open another terminal to send 3 dynamic requests to occupy all the threads. Then send another 2 static requests. If it works well, the 2 static requests will be blocked until the dynamic requests finish.

4.1.3 within_working_dir_test.sh

This bash file is made to test the situation where the request wants to refer to files outside of this sub-tree. First, open a new terminal and start the server which has 3 threads and 3 buffers. Then send a request which refers to a spin.cgi within the sub-tree. Then send a request which wants to refer to a hello_world.html outside of the sub-tree. If it works well, the dynamic request will be successfully handled by the server. And for the static request, there will be a 403 Forbidden response.

4.2 Not implemented

4.2.1 Scheduling algorithms

In this project we where asked to implement a multi-threaded server which made imperative the selection of a scheduling algorithm, in order to handle the requests that were on the buffer. In our solution a **First In, First Out (FIFO)** (also called First Come, First Served FCFS) algorithm was implemented for simplicity reasons. However, we cannot deny that they are many more optimal (and complex) scheduling algorithms that would make the server run in a more efficient way, in terms of s response time, turnaround time, and total wait time [2], such as:

- Shortest Job First (SJF)
- Shortest Time-to-Completion First (STCF)
- Round-Robin (RR)

5 Conclusion

The development of this project gave a further understanding of the concepts treated in class such as threads, locks and the consumer-producer relationship, among others.

This project gave us the opportunity to practice reverse engineering in the way that given a working code, we had to find out how the given functions in the code were built, what did they do, and they were related to another pieces of the same code in order to understand the server as a whole and be able to implement the features that were asked.

We managed to learn with this project the basic architecture of a simple web server and the importance of managing synchronization in an application as the producer-consumer relationship, while implementing concurrency.

Undeniably, the security aspect was of high importance in this project. In our simple server it was imperative to not let the server running beyond testing as it is a potential backdoor into the files of our system. A simple security consideration was made, but this only takes into account the pathname, leaving open other possible security threats that could be hiding inside the content of the request, among others. This approach made us take into account the security aspect for future projects.

References

[1] P. JOHNSTON. Creating a Circular Buffer in C and C++. [Online] Available: https://embeddedartistry.com/blog/2017/05/17/creating-a-circular-buffer-in-c-and-c/

[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Operating Systems: Three Easy Pieces. Arpaci-Dusseau Books. August, 2018 (Version 1.00)