Computer Operating Systems

BLG 312E

Homework 3 Report

KAAN KARATAŞ
karatask20@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 31.05.2024

CONTENTS

1. Introduction

The goal of this assignment is to enhance the basic web server provided by implementing a multi-threaded architecture. Single-threaded servers face performance issues as they can only handle one HTTP request at a time, causing delays for subsequent clients. By introducing multi-threading, we aim to overcome this limitation and improve server responsiveness.

In this project, we will create a fixed-size pool of worker threads to handle incoming HTTP requests concurrently. The master thread will be responsible for accepting connections and placing them into a buffer, while worker threads will process these requests. Through the use of synchronization mechanisms such as semaphores, we will ensure proper coordination between threads, enabling efficient handling of multiple requests simultaneously.

2. Implementation

2.1. Structs

The connection_t structure is designed to store information about client connections, specifically holding the connection file descriptor (connfd) and the client's address (clientaddr). This structure is essential for managing the details of each incoming connection. On the other hand, the worker_thread_t structure is used to manage worker threads, containing a pthread_t type thread and an integer id to uniquely identify each thread. These structures facilitate the server's ability to handle multiple connections concurrently by associating each connection with a thread that processes requests.

Algorithm 1 Connection Structure

```
1: Structure connection_t:
2:     Integer connfd
3:     Struct sockaddr_in clientaddr
```

Algorithm 2 Worker Thread Structure

```
1: Structure worker_thread_t:
2:     Thread pthread_t thread
3:     Integer id
```

2.2. Variables

The server starts by defining several global variables and semaphores for managing connection buffers. The buffer_size defines the size of the connection buffer, which is an array holding up to MAX_BUFFERS connections. buffer_in and buffer_out are indices used to track where new connections are added and where existing connections are processed, respectively. Semaphores mutex, full, and empty are used to ensure thread-safe operations on the buffer, preventing race conditions and ensuring that the buffer is accessed in a controlled manner.

Algorithm 3 Global Variables and Constants

```
1: MAX_BUFFERS ← 1024;                          ▷ Maximum number of buffers
2: Integerbuffer_size;                          ▷ Size of the buffer
3: Integerbuffer[MAX_BUFFERS];                  ▷ Buffer for connections
4: Integerbuffer_in ← 0;                        ▷ Index for the next connection to be inserted
5: Integerbuffer_out ← 0;                       ▷ Index for the next connection to be taken
6: sem_tmutex;                                  ▷ Binary semaphore for locking
7: sem_tfull;                                   ▷ Counting semaphore for full
8: sem_tempty;                                  ▷ Counting semaphore for empty
9: worker_thread_t * worker_threads;            ▷ Array of worker threads
```

2.3. Get Args

The getargs function is designed to parse command line arguments to retrieve the port number for the server to listen on. It accepts three parameters: a pointer to an integer port, an integer argc representing the number of command line arguments, and an array of strings argv containing the arguments. The function first checks if the number of arguments (argc) is exactly 2. If not, it prints an error message indicating the correct usage and terminates the program using exit(1). If the correct number of arguments is provided, it converts the second argument (argv[1]) from a string to an integer using the atoi function and assigns this value to the location pointed to by port. This ensures that the server has the port number needed to start listening for incoming connections.

Algorithm 4 Get Arguments Function

```
1: procedure getargs(*port, argc, *argv[])
2:     if argc ≠ 2 then
3:         fprintf(stderr, "Usage: %s <port>\n", argv[0]);
4:         exit(1);
5:     end if
6:     *port ← atoi(argv[1]);
7: end procedure
```

2.4. Initialize Thread Pool

The initialize_thread_pool function sets up the server's thread pool by first allocating memory for an array of worker_thread_t structures based on num_threads. It then initializes three semaphores: mutex for mutual exclusion (starting at 1), full for tracking filled buffer slots (starting at 0), and empty for tracking empty buffer slots (starting at buffer_size). In a loop, it assigns an ID to each thread and creates the thread using pthread_create, passing the worker function and a pointer to the corresponding worker_thread_t structure. This ensures all worker threads are initialized and ready to process connections.

Algorithm 5 Initialize Thread Pool Function

```
1: procedure initialize_thread_pool(num_threads)
2:     worker_threads ← malloc(sizeof(worker_thread_t) × num_threads); ▷ Allocate memory for worker threads
3:     sem_init(&mutex, 0, 1);                                  ▷ Initialize mutex semaphore
4:     sem_init(&full, 0, 0);                                   ▷ Initialize full semaphore
5:     sem_init(&empty, 0, buffer_size);                        ▷ Initialize empty semaphore
6:     for each i in [0, num_threads) do
7:         worker_threads[i].id ← i;                            ▷ Set worker thread ID
8:         pthread_create(&worker_threads[i].thread, NULL, worker, &worker_threads[i]); ▷ Create worker thread
9:     end for
10: end procedure
```

2.5. Destroy Thread Pool

The ‘destroy_thread_pool‘ function is responsible for cleaning up resources allocated for the thread pool. It iterates through each worker thread in the pool and waits for them to terminate using ‘pthread_join‘. Once all threads have terminated, it destroys the semaphores (‘mutex‘, ‘full‘, and ‘empty‘) using ‘sem_destroy‘. Finally, it frees the memory allocated for the array of worker threads using ‘free‘. This ensures proper cleanup and prevents resource leaks after the server has finished running.

Algorithm 6 Destroy Thread Pool Function

```
1: procedure destroy_thread_pool(num_threads)
2:     for each i in [0, num_threads) do
3:         pthread_join(worker_threads[i].thread, NULL);        ▷ Join worker threads
4:     end for
5:     sem_destroy(&mutex);                                     ▷ Destroy mutex semaphore
6:     sem_destroy(&full);                                      ▷ Destroy full semaphore
7:     sem_destroy(&empty);                                     ▷ Destroy empty semaphore
8:     free(worker_threads);                                    ▷ Free memory allocated for worker threads
9: end procedure
```

2.6. Worker

The worker function represents the worker thread’s main routine. It continuously loops to handle incoming connections. Within the loop, it waits for the full semaphore to indicate that there’s at least one connection in the buffer. Once available, it acquires the mutex semaphore to ensure exclusive access to shared resources. Then, it retrieves a connection file descriptor from the buffer, updates the buffer index accordingly, releases the mutex semaphore, and signals that an empty slot is available in the buffer using the empty semaphore.

Algorithm 7 Worker Thread Function

```
1: function worker(arg)
2:     thread ← (worker_thread_t*)arg;                    ▷ Get worker thread
3:     while true do                                      ▷ Infinite loop
4:         sem_wait(&full);                               ▷ Wait for full semaphore
5:         sem_wait(&mutex);                              ▷ Wait for mutex semaphore
6:         connfd ← buffer[buffer_out];                   ▷ Get connection file descriptor
7:         buffer_out ← (buffer_out + 1)%buffer_size;     ▷ Update buffer_out
8:         sem_post(&mutex);                              ▷ Release mutex semaphore
9:         sem_post(&empty);                              ▷ Release empty semaphore
10:        requestHandle(connfd);                         ▷ Handle the request
11:        Close(connfd);                                 ▷ Close the connection
12:    end while
13:    return NULL;
14: end function
```

Subsequently, it processes the received connection by calling requestHandle to handle the request and then closes the connection using Close. This loop continues indefinitely, ensuring that the worker thread remains active and responsive to incoming connections. Finally, the function returns NULL once the loop exits, though this return statement is unreachable due to the infinite loop.

2.7. Main

The ‘main‘ function serves as the entry point for the server application. It begins by checking if the correct number of command-line arguments (5) are provided. If not, it displays a message indicating the required arguments and exits with an error.

Next, it converts the command-line arguments to their respective types: ‘port‘ as the port number for the server to listen on, ‘num_threads‘ as the number of threads in the thread pool, ‘buffer_size‘ as the size of the buffer, and ‘schedalg‘ as the scheduling algorithm (though not used in this basic implementation).

After ensuring the buffer size is within the allowed limit, it opens a listening socket on the specified port using ‘Open_listenfd‘. Then, it initializes the thread pool with the specified number of threads using the ‘initialize_thread_pool‘ function.

The main server loop continuously accepts new connections from clients. For each connection accepted, it waits for the buffer to have available space (‘empty‘ semaphore), acquires the ‘mutex‘ semaphore to safely add the new connection to the buffer, updates the buffer index, and signals that the buffer is no longer empty (‘full‘ semaphore).

Once the server loop exits (which typically doesn’t happen unless there’s an error or the program is terminated externally), it cleans up the thread pool using ‘destroy_thread_pool‘ to join all worker threads and release resources. Finally, it returns 0 to indicate successful execution.

3. Usage And Compilation

To use the web server code from the terminal, follow these steps:

1. Navigate to the directory containing the source code files of the web server.
2. Open a terminal window.
3. Compile the source code using a C compiler such as GCC. For example, run the following command: *gcc -o server server.c -lpthread*

   This command compiles the server.c file and generates an executable named server. The -lpthread flag is used to link the pthread library, which is required for multi-threading support.
4. Once the compilation is successful, you can run the server by executing the generated executable. Use the following command: ./server <port> <threads> <buffers> <schedalg>

   Replace <port>, <threads>, <buffers>, and <schedalg> with the appropriate values. These parameters specify the port number for the server to listen on, the number of threads in the thread pool, the size of the buffer, and the scheduling algorithm (if applicable).
5. Once the server is running, it will start listening for incoming HTTP requests on the specified port. You can now access the server from a web browser or send HTTP requests programmatically for testing purposes.

4. Conclusion

In conclusion, this project has provided valuable insights into the challenges and solutions involved in developing a multi-threaded web server. By transitioning from a single-threaded to a multi-threaded architecture, we have significantly improved the server's performance and responsiveness to client requests.

Through the implementation of a fixed-size pool of worker threads and proper synchronization mechanisms, such as semaphores, we have effectively managed concurrent connections and mitigated potential bottlenecks. This approach allows the server to efficiently handle multiple requests simultaneously, enhancing overall scalability and user experience.

While this project focused on a basic implementation of a multi-threaded web server, the concepts and techniques learned can be extended to more complex server architectures. By continuing to explore and refine these principles, we can further optimize server performance and meet the evolving demands of modern web applications.

Overall, this project has been a valuable learning experience, providing hands-on practice with multi-threading and concurrent programming concepts.