Computer Operating System Homework 3

Mehmet Ali Balıkçı - 150200059

May 31, 2024

1 Introduction

In the assignment, we were asked to convert a single-threaded web server to multi-core and write a program that can evaluate many requests simultaneously in different threads using semaphore and buffer. In this assignment, I accomplished the desired operation by only making some changes to the codes in the server.c file, and I tested this by trying it many times in the terminal and confirmed that my program was working correctly.

2 Specifications of Program

The server.c file included in the draft files given to us included the getargs and main functions.

2.1 Getargs Function

The getargs function was just getting the port data and checking if the number of parameters was correct. I revised this function to obtain the numthreads parameter, which indicates how many threads the user wants to perform the operations with, as requested, and the buffersize parameter, which is the size of the buffer that the user decides what the size is.

```
void getargs(int *port, int *numthreads, int *buffersize, int argc, char *argv[])
{
    if (argc != 4) {
        fprintf(stderr, "Usage: %s <port> <numthreads> <buffersize>\n", argv[0]);
        exit(1);
    }
    *port = atoi(argv[1]);
    *numthreads = atoi(argv[2]);
    *buffersize = atoi(argv[3]);
}
```

Figure 1: Getargs Function

2.2 Workerthread Function

I wrote this function to fulfill user requests in different threads. First of all, it should be noted that the buffer with the size entered by the user and the ID of each worker thread I created are stored in a globally defined structure called thread_arg_t. The reason why this structure is defined globally is to ensure that each worker thread can access the buffer variable, so that it can be checked synchronously whether the buffer is full or not.

A variable from the thread_arg_t structure is defined in the Worker Thread function, and the necessary information for the current thread is taken from the void arg parameter that comes to the function. Then, within this function, the threads work in a continuous loop, but since the buffer is a common variable, a lock mechanism is used and at the same time, semaphore is used to ensure synchronization, as stated in the assignment description. Here, the file descriptor of the connection is stored in the buffer and the currently running thread can directly access this descriptor during the process.

```c
void *workerthread(void *arg) {
    thread_arg_t *thread_arg = (thread_arg_t *)arg;
    int *buffer = thread_arg->buffer;
    int thread_id = thread_arg->thread_id;

    while (1) {
        sem_wait(&full_slots);  // Wait for at least one full slot
        pthread_mutex_lock(&mutex);
        int connfd = buffer[--num_connections];
        pthread_mutex_unlock(&mutex);
        sem_post(&empty_slots); // Signal that one slot is now empty

        // Handle request
        printf("Thread %d handling request from descriptor: %d\n", thread_id, connfd);
        requestHandle(connfd);
        close(connfd);
    }
    return NULL;
}
```

Figure 2: Workerthread Function

2.3 Main Function

Finally, in the main function section, the data expected from the user is received via the terminal and sent to the necessary functions.

Here, an int buffer pointer is created according to the buffer size entered by the user. Semaphore variables are initialized. Again, the number of threads entered by the user is created. These are the worker threads mentioned in the assignment description. An array of type thread_arg_t is created to store the buffer pointer used synchronously for each thread and the id of the thread. Threads are created with the help of a for loop. While each thread is being created, the buffer of the thread and the buffer information in the variable that stores the thread id are matched with the buffer pointer created in main, thus ensuring synchronization. These were the side of fulfilling the request from the user.

```c
int main(int argc, char *argv[])
{
    int listenfd, connfd, port, clientlen, numthreads, buffersize, *buffer;
    struct sockaddr_in clientaddr;

    getargs(&port, &numthreads, &buffersize, argc, argv);
    buffer = (int*)malloc(sizeof(int) * buffersize);
    if (buffer == NULL) {
      perror("malloc");
      exit(1);
    }

    sem_init(&empty_slots, 0, buffersize);  // Initialize semaphore for empty slots
    sem_init(&full_slots, 0, 0);  // Initialize semaphore for full slots

    pthread_t workerthreads[numthreads];
    thread_arg_t thread_args[numthreads];

    // Create worker threads
    for (int i = 0; i < numthreads; i++) {
        thread_args[i].buffer = buffer;
        thread_args[i].thread_id = i;
        if (pthread_create(&workerthreads[i], NULL, workerthread, &thread_args[i]) != 0) {
            perror("pthread_create");
            free(buffer);
            exit(1);
        }
    }
```

Figure 3: Create Threads

On the other hand, the port is listened to through a continuously running while loop, and the connection file descriptor of incoming requests is written to the buffer using semaphore and mutex.

```c
    listenfd = Open_listenfd(port);
    while (1) {
      clientlen = sizeof(clientaddr);
      connfd = Accept(listenfd, (SA *)&clientaddr, (socklen_t *) &clientlen);

      sem_wait(&empty_slots);  // Wait for at least one empty slot
      pthread_mutex_lock(&mutex);
      buffer[num_connections++] = connfd;
      pthread_mutex_unlock(&mutex);
      sem_post(&full_slots);  // Signal that one slot is now full
    }

    free(buffer);
    sem_destroy(&empty_slots);  // Destroy semaphore for empty slots
    sem_destroy(&full_slots);  // Destroy semaphore for full slots
    return 0;
}
```

Figure 4: Listen to the Port

3 Sample Printout

```
Thread 2 handling request from descriptor: 6
GET /1.jpg HTTP/1.1
HTTP/1.0 404 Not found
Content-Type: text/html
Content-Length: 149

<html><title>blg312e Error</title><body bgcolor=fffff>
404: Not found
<p>blg312e Server could not find this file: ./1.jpg
<hr>blg312e Web Server
GET /1.png HTTP/1.1
Thread 3 handling request from descriptor: 4
GET /2.jpg HTTP/1.1
Thread 1 handling request from descriptor: 5
```

Figure 5: Sample Output