Computer Operating Systems

BLG312E

Homework 3 Report

Ömer YILDIZ
yildizom20@itu.edu.tr

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

1. Multithreaded WebServer

1.1. Introduction

The core idea behind a concurrent server is to use the producer-consumer problem, where each client request acts as a new job, represented by a file descriptor. Worker threads act as consumers, processing jobs from a buffer according to a specified scheduling policy. This approach ensures that the main thread can continually listen for new requests without being blocked by request handling.

1.2. Architecture

The server architecture consists of the following components:

- **Master Thread**: Accepts connections for the incoming HTTP requests in queue.Schedule the requests in the queue according the scheduling policy.
- **Worker Threads**: Fetch requests from the buffer and process them.

```
void* thread_worker(void* arg)
{
    thread_arg* args = (thread_arg*)arg;
    while(1)
    {
    // Getting the request that this thread has to handle from the scheduler
    int conn_fd = get_from_scheduler(args->workers, args->scheduler);
    printf("\n");
    printf("*********************************************\n");
    printf("\n");
    printf("thread = \%d, conn_fd = \%d\n", args->num_request, conn_fd);
    printf("\n");
    printf("*********************************************\n");
    printf("\n");
    // Handle request
    requestHandle(conn_fd);

    // Close connection
    Close(conn_fd);
    }
    return NULL;
}
```

- **Scheduler**: Manages the buffer and schedules requests based on the selected policy (FIFO, SFF, RFF). The scheduler structure, `scheduler_t`, is defined as follows:

```
typedef struct __scheduler_t {
    char* policy;       % Scheduling policy (e.g., FIFO, SFF, RFF)
    int buffer_size;    % Maximum size of the buffer
    int curr_size;      % Current size of the buffer
    heap* Heap;         % Pointer to a heap data structure
    queue* Queue;       % Pointer to a queue data structure
} scheduler;
```

1.3. Scheduling API

The basic server client relationship :

- **The Scheduling Policy API:** Client makes an HTTP request to the server (master thread), this main thread on receive this request puts the request in the buffer data structure and goes immediately to listen for more request, hence the main thread does not block other coming requests. Also, at the time of launching the server, it creates worker threads which pick up the request from this buffer data structure where each request is expressed as a file descriptor (integer) for that client. These worker threads upon picking up request start serving the requested file or the cgi program.

1.4. Client

I designed the client program to send multiple requests concurrently to the server. Each request is sent from a separate thread, allowing for simultaneous connections to the server.

```
// Total number of files given as input from the command line
  int concur-clients = argc - 3;
  pthread-t threads[concur-clients];

  // Creating threads to server requests for all the files concurrently
  for(int i = 0; i < concur-clients; i++) {

        client_data *d = malloc(sizeof(client_data));
        if(d == NULL) {
            printf("Could not create request for: %s\n", argv[3 + i]);
            continue;
        }
        d->host = host;
        d->port = port;
        d->filename = argv[3 + i];
        pthread_create(&threads[i], NULL, single_client, (void *)d);
        //single_client function open a single
        //connection to the specified host and port
    }
    for(int i = 0; i < concur-clients; i++) {
        pthread_join(threads[i], NULL);
    }
```

1.5. Server

The server supports different scheduling policies to manage how requests are handled:

1.5.1. FIFO (First In, First Out)

Worker threads handle requests in the order they arrive. This policy ensures fairness but does not guarantee the fastest overall processing time.I used queuq data structure and implement basic queue operations for the FIFO model // Queue structure

```
typedef struct __queue_t {
    int max_size;
    int curr_size;
    int fill;
    int use;
    node* array;
} queue;

// Methods for Queue
queue* init_queue(int queue_size);
void insert_in_queue(int conn_fd, queue* Queue);
int get_from_queue(queue* Queue);
```

1.5.2. SFF (Shortest File First)

Worker threads handle the request for the smallest file first. This approach approximates Shortest Job First scheduling, potentially reducing average response time but risking starvation for large requests.I used heap data structure to get smallest file for the every request

```
// Heap structure
typedef struct __heap_t {
    int curr_size;
    int max_size;
    node* array;
} heap;

// Methods for Heap
    void _swap(node* x, node* y);
    heap* init_heap(int heap_size);
    void insert_in_heap(int conn_fd, off_t parameter, heap* Heap);
    void heapify(heap* Heap, int index);
    int extract_min(heap* Heap);
    int heap_comparator(heap* Heap, int i, int j);
```

1.5.3. RFF (Recent File First)

Worker threads handle the request for the file with the most recent modification time. This policy might be useful in scenarios where recently updated content is prioritized.I used heap data structure and file-properties structure to follow last modification time .Heap created according to last modification time. The helper structure for file properties, `file_prop_t`, is defined as follows:

```
typedef struct __file_prop_t {
    char* file_name;    % Name of the file
    off_t file_size;    % Size of the file
    time_t mod_time;    % Last modification time of the file
} file_prop;
```

1.6. Implemantation

1.6.1. Server Initialization and Main Loop

The `server.c` file initializes the web server. It parses command-line arguments to determine the port number, number of threads, buffer size, and scheduling algorithm. The main function sets up a scheduler and a thread pool, then listens for incoming connections on the specified port. When a connection is accepted, it passes the socket descriptor to the scheduler for handling.

1.6.2. Scheduler Initialization and Scheduling

Within `thread_pool.c`, the scheduler is initialized based on the scheduling policy (`SFF`, `FIFO`, or `RFF`). Depending on the policy, requests are managed in either a priority queue or a heap data structure. The scheduler tracks the current size of requests and determines the next request to be processed based on the policy.

1.6.3. Thread Pool Initialization and Worker Threads

The thread pool is initialized in `thread_pool.c`, creating a specified number of worker threads. Synchronization mechanisms like mutexes and semaphores are set up for thread coordination. Each worker thread runs the `thread_worker` function, which retrieves requests from the scheduler and handles them using the `requestHandle()` function.

1.6.4. Thread Worker Handling Requests

The `thread_worker` function in `thread_worker.c` represents a worker thread. It continuously fetches requests from the scheduler and processes them. Upon receiving a socket descriptor from the scheduler, it calls `requestHandle()` to handle the HTTP request, generating appropriate responses. After processing, the connection is closed, and the thread repeats the process.

1.7. Conclusion

This concurrent multithreading web server demonstrates the effectiveness of different scheduling policies in managing client requests. By leveraging multiple worker threads and a well-defined scheduling strategy, the server can handle numerous concurrent connections efficiently but I observed that RFF underperformed compared to FIFO and SFF.I think the biggest reason is when the modification time gets very high heap data structure get slower.