Multi-Threaded Web Server

Yewon Lee

1 Design of Shared Data Structure and Synchronization

The shared data structure used in the web server program comprises a buffer (`request`) responsible for holding incoming client requests. This buffer is accessed concurrently by multiple worker threads tasked with processing these requests. To ensure seamless coordination and prevent data corruption in this shared environment, the program employs a combination of mutex locks and semaphores.

A mutex lock (`mutex`) serves as the primary synchronization mechanism, facilitating mutual exclusion while accessing the shared buffer. Before reading from or writing to the buffer, a thread acquires the mutex lock, ensuring exclusive access and preventing other threads from modifying the buffer concurrently. This helps maintain data integrity by ensuring that only one thread operates on the buffer at any given time.

In addition to mutex locks, semaphores (`sem_empty` and `sem_full`) are utilized to control access to the buffer based on its current state. The `sem_empty` semaphore counts the number of available slots in the buffer, while `sem_full` counts the number of filled slots. Threads interacting with the buffer use these semaphores to coordinate their actions: when adding a request, a thread decrements `sem_empty` and waits if the buffer is full; when processing a request, it decrements `sem_full` and waits if the buffer is empty. This mechanism ensures that threads operate on the buffer in a synchronized manner, preventing issues such as race conditions and deadlocks.

By combining mutex locks for mutual exclusion and semaphores for coordination, the web server program effectively manages concurrent access to the shared buffer, guaranteeing data integrity and preventing synchronization problems in a multi-threaded environment.

2 List of Accomplishments and Testing Procedures

In the development of the web server program, several key accomplishments were achieved to meet the project requirements. These include the implementation of a multi-threaded architecture capable of handling concurrent client requests, integration of crash handling functionality to dynamically replace crashed threads and maintain uninterrupted service, design and implementation of a robust shared data structure and synchronization mechanism to ensure thread safety and prevent data corruption, and validation of the program's functionality through comprehensive testing procedures to identify and address any potential concurrency issues or synchronization problems.

To ensure the reliability and correctness of the web server program, thorough testing procedures were conducted. These procedures involved unit testing of individual components to verify their functionality in isolation, integration testing to assess the interaction between different modules and ensure seamless operation of the entire system, stress testing under various workload scenarios to evaluate the program's performance and scalability, testing for concurrency issues including race conditions and deadlocks to identify and resolve any synchronization problems, and validation of crash handling functionality to verify the program's ability to recover from thread failures and maintain uninterrupted service.

3 Crash Handling

Crash handling in the web server program is achieved through a combination of thread management and error detection mechanisms. The program employs a pool of worker threads responsible for processing incoming client requests. Each worker thread continuously listens for requests, processes them, and returns the response to the client.

To handle crashes, the program periodically checks the status of worker threads using the `pthread_tryjoin_np()` function, which allows for non-blocking joins with worker threads. If a worker thread has crashed, the `pthread_tryjoin_np()` function returns a value indicating that the thread has terminated. Upon detecting a crashed thread, the program dynamically replaces it by creating a new worker thread to maintain the desired number of active threads in the thread pool.

Additionally, the program incorporates a crash rate parameter specified by the user via command-line arguments. This parameter determines the likelihood of a thread crashing abnormally during execution. The crash rate is expressed as a percentage, with higher values indicating a greater probability of thread crashes.

By adjusting the crash rate, users can simulate different failure scenarios and evaluate the robustness of the program's crash handling mechanism.

Through this approach, the web server program ensures the continuous operation of the server even in the event of thread failures. By dynamically replacing crashed threads, the program maintains uninterrupted service, minimizing downtime and ensuring reliable performance under varying conditions.

4 Usage

Compiling the program can be done using the following command in the terminal:

```
make
```

Once the program is compiled, executing the program can be done using the following command:

```
./webserver <port_number> <#_of_threads> <crash_rate>
```

`<port_number>` Specifies the port number within the range (2001 to 49999).

`<#_of_threads>` Optional parameter indicating the number of threads (default is 10). If less than 1, set as 1. If greater than 10, set as 10.

`<crash_rate>` Optional parameter specifying the crash rate percentage (default is 0). If less than 0, set as 0. If greater than 50, set as 50.

and

```
./client <ip_address> <port_number> <#_of_threads>
```

`<ip_address>` Specifies the IP address.

`<port_number>` Specifies the port number within the range (2001 to 49999).

`<#_of_threads>` Optional parameter indicating the number of threads (default is 10). If less than 1, set as 1. If greater than 10, set as 10.

5 Example Output

The following command processes the web server program on port 4000 using 10 working threads with a 0% crash rate:

```
./webserver 4000 10 0
```

The following command processes the client program on port 4000 at 127.0.0.1 using 10 working threads:

```
./client 127.0.0.1 4000 10
```

Upon completion, the program provides detailed output of each working threads with each request information. Additionally, the client program provides the amount of time taken to process.

Output:

```
./webserver 4000 10 0
CRASH RATE = 0%
HTTP server listening on port 4000
[pid 12531, tid 6] Received a request from 127.0.0.1:34345
[pid 12531, tid 6] (from 127.0.0.1:34345) URL: GET /index.html HTTP/1.0
[pid 12531, tid 2] Received a request from 127.0.0.1:34348
[pid 12531, tid 2] (from 127.0.0.1:34348) URL: GET /index.html HTTP/1.0
[pid 12531, tid 5] Received a request from 127.0.0.1:34344
[pid 12531, tid 5] (from 127.0.0.1:34344) URL: GET /index.html HTTP/1.0
[pid 12531, tid 6] Reply: filesend /home/myid/yl48687/CSCI4730/p2/index.html
[pid 12531, tid 2] Reply: filesend /home/myid/yl48687/CSCI4730/p2/index.html
[pid 12531, tid 5] Reply: filesend /home/myid/yl48687/CSCI4730/p2/index.html
[pid 12531, tid 8] Received a request from 127.0.0.1:34350
[pid 12531, tid 8] (from 127.0.0.1:34350) URL: GET /index.html HTTP/1.0
[pid 12531, tid 8] Reply: filesend /home/myid/yl48687/CSCI4730/p2/index.html
[pid 12531, tid 11] Received a request from 127.0.0.1:34358
[pid 12531, tid 11] (from 127.0.0.1:34358) URL: GET /index.html HTTP/1.0
[pid 12531, tid 9] Received a request from 127.0.0.1:34356
[pid 12531, tid 9] (from 127.0.0.1:34356) URL: GET /index.html HTTP/1.0
[pid 12531, tid 7] Received a request from 127.0.0.1:34362
[pid 12531, tid 7] (from 127.0.0.1:34362) URL: GET /index.html HTTP/1.0
[pid 12531, tid 11] Reply: filesend /home/myid/yl48687/CSCI4730/p2/index.html
[pid 12531, tid 9] Reply: filesend /home/myid/yl48687/CSCI4730/p2/index.html
[pid 12531, tid 10] Received a request from 127.0.0.1:34354
[pid 12531, tid 10] (from 127.0.0.1:34354) URL: GET /index.html HTTP/1.0
[pid 12531, tid 7] Reply: filesend /home/myid/yl48687/CSCI4730/p2/index.html
[pid 12531, tid 10] Reply: filesend /home/myid/yl48687/CSCI4730/p2/index.html
[pid 12531, tid 12] Received a request from 127.0.0.1:34352
[pid 12531, tid 12] (from 127.0.0.1:34352) URL: GET /index.html HTTP/1.0
[pid 12531, tid 13] Received a request from 127.0.0.1:34360
[pid 12531, tid 13] (from 127.0.0.1:34360) URL: GET /index.html HTTP/1.0
[pid 12531, tid 12] Reply: filesend /home/myid/yl48687/CSCI4730/p2/index.html
[pid 12531, tid 13] Reply: filesend /home/myid/yl48687/CSCI4730/p2/index.html


./client 127.0.0.1 4000 10
Request: GET 127.0.0.1:4000//index.html, # of client: 10
[tid 14965] received 1458269 bytes (0.108781 sec).
[tid 14966] received 1458269 bytes (0.109328 sec).
```

```
[tid 14970] received 1458269 bytes (0.111080 sec).
[tid 14969] received 1458269 bytes (0.110858 sec).
[tid 14967] received 1458269 bytes (0.127373 sec).
[tid 14984] received 1458269 bytes (0.102453 sec).
[tid 14968] received 1458269 bytes (0.126631 sec).
[tid 14978] received 1458269 bytes (0.109351 sec).
[tid 14982] received 1458269 bytes (0.109046 sec).
[tid 14971] received 1458269 bytes (0.129895 sec).
Time to handle 10 requests (0 failed): 0.142428 sec
```