Project 3 - Concurrent HTTP Server(CS406:Spr 2025)
Crystal Yeung and Leah Boyle
Github repository: https://github.com/crystalcy16/webserver

Contents:

Description implementation:

The project implements a multi-threaded web server consisting of a master thread and N worker threads from a thread pool. It has an HTTP client that can send and print responses. It supports both static and dynamic requests. There is FIFO(first in first out) and SFF(shortest file first) scheduling. The default is FIFO.

Multithreading allows the server to handle multiple client requests in parallel by using a fixed thread pool designated by -t and a fixed buffer designated by -b. N number of worker threads are created and take requests from the buffer. Mutexes and condition variables are used to make sure the buffer isn't full or empty.

FIFO is first in first out where it adds requests to the tail of the buffer. Then the buffer is dequeue by the oldest then the buffer shifts forward.

SFF simply scans the buffer to find the smallest file. The file size is checked through the entire buffer and picks the smallest size.

Security checks if url has ../filename and blocks it if it does.

Algorithms used and Implementation:

For multi-threading, the main function takes in the parameters. It checks the directory and the port for access. Then -t gets the number of threads in the thread pool for the worker threads while -b is the size of the buffer where the requests are queued. -s checks what schedule it is and default is FIFO.

Each worker thread runs the worker function, an infinite loop. It waits for work to be available, dequeues the request from the buffer and handles the http request and close the connection. It also signals to the main thread if there is room.

The master thread in main accpets the requests and places them into the circular buffer using enqueue. Then depending on the schedule either dequeue_fifo or SFF sends them to the worker threads. There are also fail-safes in the buffer. Using mutex and conditionals, when the buffer is full, the main thread waits. When buffer is empty, the worker threads wait. pthread_cond_signal() is used to wake up the worker threads if buffer has requests.

A circular buffer was used since I tried shifting elements but it often broke and had free errors. It also makes the buffer a fixed-sized as requested. After reaching the end of the array, it wraps back to index 0

Both circular buffers are used in scheduling.

In FIFO, it puts the request into the buffer's tail and increments tail. Then a worker thread grabs the oldest request from the head of the buffer where the head points to the head points to the first request that arrived. Then the buffer will head move forward (wrapping if needed). Hence it guarantees that first in will always be first out.

SFF enqueue is the same as FIFO since they use the same circular buffer however dequeue is different. The server scans the queue and picks the one with the smallest file size. Once -s is SFF, it checks the entire queue to see who has the smallest size using a for loop and stat. Each request compares file size and the best or smallest file size's index is kept. Then the worker thread runs the index with the best file. To make sure there is no empty space in the buffer, the tail is copied into the place of the best index to replace it and the tail is adjusted accordingly.

In the main() function it checked if incoming http had .. and if so, it returned 404 for the security aspect.

Verification Plan and Demonstration:

To achieve the core requirements on the client side, a script was used to test for dynamic and static requests on the client's end. Spin.cgi was the dynamic and 3 html files of varying sizes (small, medium, and large) were used as dynamic.

Tests for clients side(./tests.sh):

1. Multi-threading (static)

Under Server: you can see the different thread numbers handling each request (denoted by get…) Queue is just placing the requests in the queue to run. As seen, these static files are run under different threads hence it is a working multi-thread.

```
-----[TEST1.1] MULTI-THREADING (static)-----
***Starting server with NONE scheduling (6 threads)***
QUEUE: Enqueuing fd=4, size=0 (NONE mode)
QUEUE: Enqueuing fd=5, size=0 (NONE mode)
SERVER: Thread 139928631850560 handling NONE (fd=4, size=0)
method:GET uri:/small.html version:HTTP/1.1
SERVER: Thread 139928640243264 handling NONE (fd=5, size=0)
method:GET uri:/small.html version:HTTP/1.1
QUEUE: Enqueuing fd=6, size=0 (NONE mode)
SERVER: Thread 139928623457856 handling NONE (fd=6, size=0)
method:GET uri:/medium.html version:HTTP/1.1
QUEUE: Enqueuing fd=4, size=0 (NONE mode)
SERVER: Thread 139928615065152 handling NONE (fd=4, size=0)
method:GET uri:/large.html version:HTTP/1.1
INFO: All curl jobs completed for TEST1.1

***EXPECTED RESULT: should be diff numbered [THREADS] ***
```

2. Multi-threading (dynamic)

For dynamic, same idea. You can see that there are multiple threads and that spin.cgi runs. (None defaults to FIFO but multi-threading doesn't show it properly)

```
-----[TEST1.2] MULTI-THREADING (dynamic)-----
***Starting server with NONE scheduling (3 threads)***
QUEUE: Enqueuing fd=4, size=0 (NONE mode)
SERVER: Thread 139803307701824 handling NONE (fd=4, size=0)
method:GET uri:/spin.cgi?1 version:HTTP/1.1
QUEUE: Enqueuing fd=5, size=0 (NONE mode)
SERVER: Thread 139803316094528 handling NONE (fd=5, size=0)
method:GET uri:/spin.cgi?3 version:HTTP/1.1
QUEUE: Enqueuing fd=6, size=0 (NONE mode)
SERVER: Thread 139803299309120 handling NONE (fd=6, size=0)
method:GET uri:/spin.cgi?2 version:HTTP/1.1
<p>Welcome to the CGI program (1)</p>
<p>My only purpose is to waste time on the server!</p>
<p>I spun for 1.00 seconds</p>
<p>Welcome to the CGI program (2)</p>
<p>My only purpose is to waste time on the server!</p>
<p>I spun for 2.00 seconds</p>
<p>Welcome to the CGI program (3)</p>
<p>My only purpose is to waste time on the server!</p>
<p>I spun for 3.00 seconds</p>
INFO All curl jobs completed for TEST1.2

***EXPECTED RESULT: should be diff numbered [THREADS] ***
```

3. FIFO

For FIFO, you can see that it enqueues the request then immediately handles it hence the first in is always the first out in order to medium, large, small.

```
-----[TEST2] FIFO ORDER-----
***Starting server with FIFO scheduling (1 threads)***
QUEUE: Enqueuing fd=4, size=0 (FIFO mode)
SERVER: Thread 140157320750656 handling FIFO (fd=4, size=0)
method:GET uri:/medium.html version:HTTP/1.1
QUEUE: Enqueuing fd=5, size=0 (FIFO mode)
SERVER: Thread 140157320750656 handling FIFO (fd=5, size=0)
method:GET uri:/small.html version:HTTP/1.1
QUEUE: Enqueuing fd=4, size=0 (FIFO mode)
SERVER: Thread 140157320750656 handling FIFO (fd=4, size=0)
method:GET uri:/large.html version:HTTP/1.1
INFO: All curl jobs completed for TEST2

***EXPECTED RESULT: should [SERVER] handled  medium, small, large ***
```

4. SFF

The order that SFF handles is the smallest size first. The launch order is large, medium,and small but as seen, the SFF handles small, medium then large.

```
-----[TEST3] SFF ORDER-----
***Starting server with SFF scheduling (1 threads)***
QUEUE: Enqueuing fd=4, size=46 (SFF mode)
QUEUE: Enqueuing fd=5, size=5052 (SFF mode)
SERVER: Thread 140108837729856 handling SFF (fd=4, size=46)
QUEUE: Enqueuing fd=6, size=553 (SFF mode)
method:GET uri:/small.html version:HTTP/1.1
SERVER: Thread 140108837729856 handling SFF (fd=6, size=553)
method:GET uri:/medium.html version:HTTP/1.1
SERVER: Thread 140108837729856 handling SFF (fd=5, size=5052)
method:GET uri:/large.html version:HTTP/1.1
INFO: All curl jobs completed for TEST3

***EXPECTED RESULT: should [SERVER] handled small, medium, large ***
```

5. Buffer full

This is 1 thread with a size 8 buffer. As seen, the thread takes the first request and the other 8 fill up the buffer. The request after the 8th one that fills up the buffer is blocked. Once a spot is free, the buffer allows the request to be taken but blocks it once again since there is another request after the full buffer.

```
-----[TEST4] MASTER BLOCKING (buffer full)-----
***Starting server with NONE scheduling (1 threads)***
QUEUE: Enqueuing fd=4, size=0 (NONE mode)
SERVER: Thread 140401935246912 handling NONE (fd=4, size=0)
method:GET uri:/spin.cgi?2 version:HTTP/1.1
QUEUE: Enqueuing fd=5, size=0 (NONE mode)
QUEUE: Enqueuing fd=6, size=0 (NONE mode)
QUEUE: Enqueuing fd=7, size=0 (NONE mode)
QUEUE: Enqueuing fd=8, size=0 (NONE mode)
QUEUE: Enqueuing fd=9, size=0 (NONE mode)
QUEUE: Enqueuing fd=10, size=0 (NONE mode)
QUEUE: Enqueuing fd=11, size=0 (NONE mode)
QUEUE: Enqueuing fd=12, size=0 (NONE mode)
BLOCKED: FULL THREAD
<p>Welcome to the CGI program (2)</p>
<p>My only purpose is to waste time on the server!</p>
<p>I spun for 2.00 seconds</p>
SERVER: Thread 140401935246912 handling NONE (fd=5, size=0)
QUEUE: Enqueuing fd=13, size=0 (NONE mode)
method:GET uri:/spin.cgi?1 version:HTTP/1.1
BLOCKED: FULL THREAD
<p>Welcome to the CGI program (1)</p>
<p>My only purpose is to waste time on the server!</p>
<p>I spun for 1.00 seconds</p>
SERVER: Thread 140401935246912 handling NONE (fd=6, size=0)
method:GET uri:/spin.cgi?1 version:HTTP/1.1
QUEUE: Enqueuing fd=4, size=0 (NONE mode)
<p>Welcome to the CGI program (1)</p>
<p>My only purpose is to waste time on the server!</p>
<p>I spun for 1.00 seconds</p>
SERVER: Thread 140401935246912 handling NONE (fd=7, size=0)
method:GET uri:/spin.cgi?1 version:HTTP/1.1
SERVER: Thread 140401935246912 handling NONE (fd=8, size=0)
method:GET uri:/spin.cgi?1 version:HTTP/1.1
INFO: All curl jobs completed for TEST4

***EXPECTED RESULT: should [BLOCK] show up ***
```

6. Security (..)

The server realized its trying to access the directory above hence it returned a 404 not allowing access.

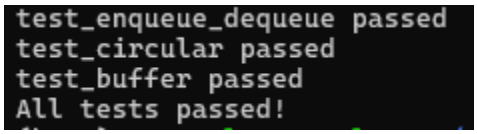
```
-----[TEST5] GOING UP IN DIRECTORY-----
***Starting server with SFF scheduling (1 threads)***
*   Trying 127.0.0.1:4444...
* Connected to localhost (127.0.0.1) port 4444 (#0)
> GET /%2e%2e/spin.cgi?1 HTTP/1.1
> Host: localhost:4444
> User-Agent: curl/7.81.0
> Accept: */*
>SECURITY: ERROR can't use [..]

QUEUE: Enqueuing fd=4, size=0 (SFF mode)
SERVER: Thread 140021830374976 handling SFF (fd=4, size=0)
method:GET uri:/%2e%2e/spin.cgi?1 version:HTTP/1.1
* Mark bundle as not supporting multiuse
* HTTP 1.0, assume close after body
< HTTP/1.0 404 Not found
< Content-Type: text/html
< Content-Length: 188
<
<!doctype html>
<head>
  <title>OSTEP WebServer Error</title>
</head>
<body>
  <h2>404: Not found</h2>
  <p>server could not find this file: ./%2e%2e/spin.cgi</p>
</body>
</html>
* Closing connection 0
INFO: All curl jobs completed for TEST4

***EXPECTED RESULT: should [SECURITY] show up ***
```

The core aspects of the implementation were tested. To make sure the enqueuing and dequeuing would work, unit testing was done. This is pivotal for the threads to work and also the scheduling itself. Also to make sure that the buffer could work circularly, without error, it also was checked that order would be preserved. Also for the buffer it checked that enqueue is blocked when buffer is full.

Unit Testing:



1. test_enqueue _dequeue: adds item to buffer then removes item from buffer. Checks both enqueue and dequeue works and buffer in empty in end.
2. Test_circular: it test the circular nature of the buffer when tail and head wrap around to the beginning.
3. test_buffer_full(): checks what enqueue does if the buffer is full (should fail)

Usage:

To run client tests:

1. make
2. ./tests.sh

*note output.txt has the outputs for the curls

To run unit tests:

1. make
2. gcc -Wall -o unittest unittest.c buffer.c
3. ./unittest

To run the server manually:

1. make
2. ./wserver -d -p 4444 -t [number desired] -b [number desired] -s [SFF or FIFO]
    a. SFF example: ./wserver -d . -p 4444 -t 1 -b 8 -s FIFO
    b. FIFO example: ./wserver -d . -p 4444 -t 1 -b 8 -s SFF
3. curl "http://localhost:4444/medium.html" & curl "http://localhost:4444/spin.cgi?3"
    a. Html files of different sizes are small.html, medium.html, large.html
    b. spin.cgi?n where n can be any number desired

Workload: (Pretty evenly)

It was even in terms of designing the project. We first made an outline and split up the work between us. Crystal made the basic skeleton for multi-threading then Leah did the scheduling and adding in security. Finally Crystal created the testing for client and helped Leah with setting up unit testing. Both Leah and Crystal worked on the report.

Implementation & testing:
Crystal: multi-threading, Client testing, Unit testing
Leah: SFF, FIFO, Security