# Design Document: Multi-Threaded HTTP Server with Logging

Julia Sales
Cruz ID: jesales

---

## 1) Goals

The goal for Assignment 2 is to modify Assignment 1, our HTTP server and add a multi-threaded HTTP server with logging. We take in the commands ex: **./httpserver -N 8 -l my_log.txt localhost 8888**. Where we have "N", which is the number of worker threads that the server uses. We also have "l", which logs the data of the requests and writes them into the file specified in the arguments. For this assignment, we must use condition variables, mutexes, and/or semaphores when implementing multi-threading.

---

## 2) Design

For this assignment, there are three parts to this assignment. The first part is modifying our code for Assignment 1 where we have to modify the PUT header, where we also take in content length. The Second part is that we have to implement multi-threading. Implementing multi-threading will be broken up into more sub parts as we have to take in the argument that counts the number of worker servers and use synchronization mechanisms in our code. The last part is that we have to also implement the logging of the requests. This will also include multiple sub parts as well.

---

### 2.1 Modify Asgn1

When we modify Asgn1, we edit one of the conditions for PUT. If a PUT request is called and there is no content length, then it is considered a bad request. We include that in the parse header function on line 16. We Also edit the target name is there is a slash at the beginning, we want to ignore the slash. We implement this in the parse header function on line 14.
Other than implementing these two things. I split up my code into three functions, and a main from Asgn1. Making my code a bit easier when I have to multithread.

Creating a sockaddr_in

```
1. if argv[1] = -N and argv[2] is not NULL then set argv[2] = nthreads
2. if argv[1] = -N and argv[2] is NULL, then set nthreads to 4
3. if argv[3] and argv[4] is not NULL, then set arg[4] to LOG_FILE
4. if argv[5] is not NULL then make it SERVER_NAME_STRING
5. if argv[6] is not NULL then make it PORT_NUMBER
6. if argv[5] is NULL print the error "Request is missing required `Host` header”
7. if argv[6] is NULL print the error "Request is missing required `Port` header”
8. struct hostent *hent = gethostbyname(SERVER_NAME_STRING /* eg
   “localhost” */);
2. struct sockaddr_in addr;
3. memcpy(&addr.sin_addr.s_addr, hent->h_addr, hent->h_length);
4. addr.sin_port = htons(PORT_NUMBER);
5. addr.sin_family = AF_INET;
```

Creating a Socket

```
1. int sock = socket(AF_INET, SOCK_STREAM, 0);
2. if no connection, when sock is 0
3.       |  Error: In socket, no connection
```

Socket Setup for Server

```
1. int enable = 1;
2. setsockopt(sock, SOL_SOCKET, SO_REUSEADOR, &enable, sizeof(enable));
3. bind(sock, (struct sockaddr *)&addr, sizeof(addr));
4.       |  if not being able to bind - Error: cannot bind
5. listen(sock, 0);
6. int cl = accept(sock, NULL, NULL);
7. if cl is < 0
8.       |  Error: Cannot accept
```

GET Function

```
1.  get_function(cl, target header <- header2)
2.      |  open the file <- file
3.      |  if file is not found <- if the open file returns -1
4.          |  HTTP/1.1 404 Not Found
5.          |  Content-length: 0
6.      |  if file is found <- if open file is not equal to -1
7.          |  read file = read(file, file buffer, buffer size);
8.          |  get Content-length (2.3b)
9.          |  HTTP/1.1 200 OK
10.         |  print out the content length
11.         |  while file size is >= size of buffer <- handling large files
12.             |  write out contents from first read
13.             |  read file again starting at file
14.         |  write remaining bits
15.     |  else
16.         |  HTTP/1.1 500 Internal Server Error
17.         |  Content-length: 0
18.     |  close file
19.     |  close read file
```

Get content length

```
1.  struct stat st;
2.  stat(target, st);
3.  content-length = st.st_size;
```

PUT Function

```
1.  put_function(cl, fd, putread, bytes_left, header2, content length, buffer size, file
    buffer)
2.      | fd = open and create the file as header2
3.      | if file cannot be created
4.          | HTTP/1.1 500 Internal Sever Error
5.          | close fd
6.      | else
7.          | if the content length is smaller than the buffer
8.              | putread = read(from cl);
9.              | write(to fd);
10.         | else
11.             | putread = read(from cl);
12.             | write(to fd);
13.             | bytes_read = content length - buffer size
14.             | while bytes_read is greater than or equal to buffer size
15.                 | putread = read from cl again
16.                 | write to fd again
17.                 | bytes_read = bytes_read - buffer size
18.             | putread = read from cl last time
19.             | write to fd with remaining bytes
20.     | HTTP/1.1 201 Created
21.     | Content-length: 0
22.     | close fd
```

Parse Header Function

```
1.  parse(cl, buffer)
2.      | char start, end
3.      | char line <- malloc(buffer size)
4.      | char outline <- malloc(buffer size)
5.      |  int line_size <- initialized to 0
6.      |  int filter_request <- initialized to 1
7.      |  header1 = GET or  PUT <- the request
8.      |  header2 = the 27 ascii string <- the file
9.      |  header3 = HTTP/1,1
10.     |  while going through the whole file
11.         |  if filter_request == 1
12.             |  parse header into header sub arrays, header1…header3
13.         |  if target name has slash at beginning then
14.             |  ignore the slash <- use memmove
15.         |  if header1 == PUT
16.             |  if no content length is found
17.                 |  ERROR flag = 1
18.                 |  HTTP/1.1 400 Bad Request
19.             |  if there is a "/" anywhere in the target name
20.                 |  ERROR flag = 1
21.                 |  HTTP/1.1 403 Forbidden
22.             |  if the target name is not 27 ascii characters
23.                 |  ERROR flag = 1
24.                 |  HTTP/1.1 400 Bad Request
25.             |  else
26.                 |  we open the file <- fd
27.                 |  int: putread, bytes_left
28.                 |  int: found_except_or_blank <- initialized to 0
29.                 |  while looping through the entire file again, finding a expect 100
30.                    request or blank line, if not found will go to end of buffer
31.                     | if Expect 100 is found <- we then know this is a curl cmd
32.                         |  send 100 continue status code
33.                         |  put_function(cl, fd, putread, bytes_left, header2, content
34.                            length, buffer size, file buffer)
35.                     |  if blank line is found  <- then we know that this is netcat
36.                         |  found_except_or_blank = 1
37.                         |  skip blank line
38.                         |  read in the next number of bytes: content_len
```

Parse Header Function (b)

```
1.          | if header1 == GET
2.              | if there is a "/" anywhere in the target name
3.                  | ERROR flag = 1
4.                  | HTTP/1.1 403 Forbidden
5.              | if the target name is not 27 ascii characters
6.                  | ERROR flag = 1
7.                  | HTTP/1.1 400 Bad Request
8.              | else
9.                  | get_function(cl, buffer)
```

---

Close the connection
*Closes at the end of the while loop, while(1)*

```
At the end of everything
close(cl);
```

---

## 2.2 Multithreading

First we have to create our threads. We have a dispatcher function that makes our threads active or waits for requests. Then we have to distribute locks and conditional variables throughout the code. Our code is thread safe because the threads will not affect our main program or change any of the variables because we assign them in each thread with a worker function. Variables are modified by the client and server. However the variable values will be different per thread, but they should not affect the same variable names in different threads. The critical region is executed where we call the worker threads.

Variables
Variables will be global so they are easier to use and access.

```
1. mutex <- global pthread mutex lock
2. empty <- global pthread conditional variable
3. full <- global pthread conditional variable
4. int n <- queue size
5. request_buff[n] <- global
6. n_available[nthreads] <- buffer that holds available threads (defined in main)
7. n_in_use = 0 <- acts as counter for queue function
8. waiting_threads = 0 <- number of threads waiting
9.  int req_written <- initialized to 0 in main
```

Creating the pthreads

This will create the number of threads that is defined at the beginning of main.

```
1.  pthread_t tid[nthreads] <- nthreads is the number of threads we request to
    make
2.  int i = 0, error
3.  while i < nthreads
4.      |  error = pthread_create(tid[i], NULL, &dispatcher, void tid[i])
5.          |  if error != 0, then thread is not created, Error: Thread cannot be
6.             created
7.          |  if error == 0, then thread has been successfully created
8.          |  i++
```

Dispatcher Function

Dispatcher is going to take the request out of the queue and assign it to a thread. It will do this until all threads are being used. If the queue is empty, the threads will wait until a request is put into the queue. This function is based off the consumer producer problem.

```
1.  void *dispatcher(void *args)
2.      |  int out = 0
3.      |  char *thrd_ptr = (char *)args
4.      |  while(1)
5.          |  mutex.acquire() <- locks section
6.          |  for i=0, i < n, i++
7.              |  if request_buff[i] == NULL
8.                  |  waiting_thread += 1
9.                  |  full.wait() <- waiting for a signal
10.                 |  waiting_thread -= 1 <- queue is full, we can now stop waiting
11.         |  while request_buff[out] == NULL
12.             |  out = (out + 1) % n
13.         |  thrd_ptr = request_buff[out]
14.         |  n_in_use -= 1
15.         |  empty.signal() <- signal that spot is empty and needs to be filled
16.         |  mutex.release() <- unlocks section
```

Queue Function

The Queue function is going to take the request it receives from the clients and put it into a queue. If the queue is full, it will wait or sleep. This function is based off the consumer producer problem.

```
1.  void *queue(int cl, char *buffer[])
2.      | int in = 0
3.      | takes in buffer from cl <- this contains all the contents of the file and will be
4.        some sort of item.
5.      | while (1)
6.          | mutex.acquire()
7.          | if n_in_use = n
8.              | empty.wait() <- stops program from putting in requests
9.          | while (req_written == 0)
10.             | while request_buff[in] != NULL <- meaning spot has a request
11.                 | in = (in + 1) % n
12.             | request_buff[in] = buffer <- use memmove
13.             | n_in_use += 1
14.             | for i = 1 to waiting_threads
15.                 | full.signal() <- tells threads that there is a request in queue
16.             | req_written = 1
17.         | mutex.release()
```

---

## 2.3 Logging

Logging will write the request, the target file, the content length, and the contents of the file in hex, into a separate log file. Multiple requests will be writing to this file so it is considered a shared resource. Logging contiguously should work because it's the same way we write to the terminal, similar from Asgn0, where we will keep writing to the log file as the log file will keep taking in information.

Writing to Log file

This block of code will be written to all the lines where a status code is printed out.

```
1. Log_buffer[] = this will contain the request, target file, and content length
2. hex_buffer[] = will contain the content length in terms of hex
3. Convert contents of file into hexadecimal values add that into lo
4. write(LOG_FILE, Log_buffer, strlen(Log_buffer)
5. write(LOG_FILE, hex_buffer, strlen(log_buffer) + strlen(hex_buffer)
```

Converting contents of file into hexadecimal values

To convert ascii into hexadecimals, we want to take the ascii value out from the buffer that stores the contents from the target files one line at a time. We then want to convert that line of ascii code into hex by going through each character and turning it into a hex value. We then want to return the newly hex converted line into the hex_buffer, which will then get called and printed to the log file.

```
1. for(i=0, j =0; i < strlen(buffer); i+=2, ++j)
2.       |  int hex = take each character from the buffer[i] and turn it into hex using ‘a’
   or ‘0’
3.       |   put hex into hex_buffer[j]

// Could implement reading large file logic and modify to convert ascii to hex line
by line instead of one byte at a time.
```