CSE130 ASGN2 DESIGN

Lang Li

May 2020

1 Goals

The goal of this program is to create a web server and a client program that can communicate with each other with no errors

2 Design

There are four parts to this program. The program first setups up the server by creating and binding the socket. Then ready to listen and accept incoming HTTP requests. After receiving the HTTP request, the program received the request, it will parse it to find out whether it is a GET or PUT request. Then the server can process the client request correspondingly.

2.1 Creating A HTTP server structure

We create a HTTP server structure by first create a structure by creating its family, port number, and the socket, then giving the every property of the structure a value to make it a functioning HTTP server that can receive requests.

Algorithm 1: Set up the server structure

```
struct sockaddr_in server_addr;
memset(server_addr, 0, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(atoi(port));
server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
socklen_t ai_addrlen = sizeof(server_addr);
int server_sockd = socket(AF_INET, SOCK_STREAM, 0);
int enable = 1;
int ret = setsockopt(server_sockd, SOL_SOCKET, SO_REUSEADDR,
  enable, sizeof(enable));
ret = bind(server_sockd, (struct sockaddr *) server_addr, ai_addrlen);
```

2.2 Listen and Accept for Client Requests

To have a HTTP server that stays on forever until someone hard shut-down the server. When accepted a request, the program will send the request along with the socket descriptor to a parser that will parse the header of the request

Algorithm 2: Listen and Accept Client Request

```
int thread_id = 0; ret = listen(server_sockd);
while true do
    int client_sockd = accept(server_sockd, client_addr, client_addrlen);
    read(client_sockd , header_buffer, buffer_size);
    char *requestType;
    sscanf(heade, "%s", requestType);
    if requestType is GET then
        sem_wait(thread_sem);
        create thread[thread_id] to call GET;
        thread_id = ++thread_id % num_threads;
    else if requestType is PUT then
        sem_wait(thread_sem);
        create thread[thread_id] to call PUT;
        thread_id = ++thread_id % num_threads;
    else if requestType is HEAD then
        sem_wait(thread_sem);
        create thread[thread_id] to call HEAD;
        thread_id = ++thread_id % num_threads;
    else
        send(bad_request) ;
    end
end
```

2.3 GET function

Get function receives the header and the socket. GET function will first store the keywords from the buffer to different key word variables.
Then it will format the file name key word because the keyword parsing will include an extra slash in front of the file name
After formatting the file name, it will check if the file name matches the valid criteria. If the filename is longer than 27 characters or contains non-ASCII character, it will be deemed invalid and refuse to process client request by sending a bad request response. I check for illegal character by using strspn to compare the file name with a string of character that contains only the valid characters

Algorithm 3: Parsing keywords from the buffer

```
sscanf(header_buffer, "%*s %s", raw file name, NULL);
sscanf(header_buffer, "%*s %*s %s", http type, NULL);
```

Algorithm 4: Adjust the file name

```
j = 0;
for $i \leftarrow 1$ to $sizeoffilename$ by 1 do
    adjusted_file_name[j] = raw_file_name[i];
    j++;
```

Algorithm 5: Check file name for valid characters

```
j = 0;
char *namingConvention = adjusted_file_name;
valid_name = true;
if namingConvention[strspn(namingConvention, validChars)] != 0
 then
    valid_name = false;
```

If the name is not valid, we respond with a bad request message, otherwise we will try to open the file in the server.

Algorithm 6: Check file name for validity

```
if file_name_length > 27 or !valid_name then
    send(bad_request);
```

If the file name cannot be open, we check why we can't open it. If error number is EACCES, then we know that we don't have permission to access the file, thus we return a forbidden response, otherwise we return not found response, since there is no other errors regarding file access we need to handle.

Algorithm 7: Check for file errors

```
int fd = open(filename);
if errno == EACCES then
    send(forbidden);
else
    send(not found);
```

If the file can be opened without errors, we send a 200 OK response.Then we start reading the file content into a buffer, and send the buffer through the client socket. Rinse and repeat the reading and sending process until we read 0 byte from file, which means we read all the contents of the file and we are done reading and sending.

Algorithm 8: Reading and Sending Data

```
char *buffer;
send(OK);
while true do
    int size_read = read(s_file, buffer, sizeof(buffer));
    if size_read == 0 then
        break;
    else
        send(client socket, buffer);
        reset(buffer);
    end
end
```

2.4 PUT function

PUT function works similar to GET in the beginning, with similar setup for the function. Parsing the header for keywords.

Algorithm 9: Parsing keywords from the buffer

```
sscanf(header_buffer, "%*s %s", raw file name, NULL);
sscanf(header_buffer, "%*s %*s %s", http type, NULL);
sscanf(header_buffer, "%*s %*s %*s %*s %*s %*s %*s %*s %*s %s",
  content_length_phrase, NULL);
sscanf(header_buffer, "%*s %*s %*s %*s %*s %*s %*s %*s %*s %*s
  %s", content_length, NULL);
```

Same procedure in adjusting the file name as GET

Algorithm 10: Adjust the file name

```
j = 0;
for $i \leftarrow 1$ to sizeoffilename by 1 do
    adjusted_file_name[j] = raw_file_name[i];
    j++;
```

If content length is not greater than 0, we know the header doesn't have an actual content length, so we set that variable for header validity later.

Algorithm 11: Parsing keywords from the buffer

```
if content_length <= 0 then
    has_content_length = false;
```

Checking for illegal characters in the file name is similar to GET

Algorithm 12: Check file name for valid characters

```
j = 0;
char *namingConvention = adjusted_file_name;
valid_name = true;
if namingConvention[strspn(namingConvention, validChars)] != 0
 then
    valid_name = false;
```

Checking if the file name is valid, we have an extra step to check the request in general, we check if the headers have content length. If not, client will receive the same treatment as not having a valid filename. Bad request respond.

Algorithm 13: Check file name for validity and content length

```
if file_name_length > 27 or !valid_name or !has_content_length then
    send(bad_request);
```

Once the request passed all our tests, we can accept and process the PUT request. We will try to open the file on our server to write client content to specified file. If the file does not exist, simply create the file and ready to write. If the file exist already, delete the old file on the server and create a fresh one to write. But if the file exist and cannot be opened, we respond with a forbidden message and stop processing this request.
200(OK)

Algorithm 14: Opening file on server

```
int fileDescriptor = open(file name, WRITE);
if fileDescriptor <= 0 then
    if errno == EACCES then
        send(forbidden);
    else
        int fileDescriptor = open(file name, CREATE | WRITE,
          PERMISSION);
    end
else
    remove(file name);
    int fileDescriptor = open(file name, CREATE | WRITE,
      PERMISSION);
end
```

If the program did not send a forbidden message we can start reading from the socket for the file content. If the file created flag is set to true, we respond with the 201 created message, otherwise, we assume we overwrote the file and send 200 OK message. If the writing process produced an error in the middle of writing, send a 500 error message.

Algorithm 15: Reading and Sending Data

```
if request is not forbidden then
    recv(client socket, fileBuffer, file length);
    write(fileDescriptor, fileBuffer, recv().length);
    if write to file failed then
        send(server error);
if created then
    send(created);
else
    send(OK);
end
```

2.5 HEAD request

HEAD request is extremely similar to GET request. The only difference is that HEAD doesn't return any file content, the only thing our server need to do is to send a response message.
We parse the keyword the same way we parse keyword in GET function.

Algorithm 16: Parsing keywords from the buffer

```
sscanf(header_buffer, "%*s %s", raw file name, NULL);
sscanf(header_buffer, "%*s %*s %s", http type, NULL);
```

We adjust the filename is the same way we did in GET.

Algorithm 17: Adjust the file name

```
j = 0;
for $i \leftarrow 1$ to $sizeoffilename$ by 1 do
    adjusted_file_name[j] = raw_file_name[i];
    j++;
```

We check the validity of the file name the same way as we do in GET, checking for file name length and illegal characters.

Algorithm 18: Check file name for valid characters

```
j = 0;
char *namingConvention = adjusted_file_name;
valid_name = true;
if namingConvention[strspn(namingConvention, validChars)] != 0
 then
    valid_name = false;
```

If the name is not valid, we respond with a bad request message, otherwise we will try to open the file in the server.

Algorithm 19: Check file name for validity

```
if file_name_length > 27 or !valid_name then
    send(bad_request);
```

If the file name cannot be open, we check why we can't open it. If error number is EACCES, then we know that we don't have permission to access the file, thus we return a forbidden response, otherwise we return not found response, since there is no other errors regarding file access we need to handle.

Algorithm 20: Check for file errors

```
int fd = open(filename);
if errno == EACCES then
    send(forbidden);
else
    send(not found);
```

If the file can be opened without errors, we send a 200 OK response.

Algorithm 21: Parsing keywords from the buffer

```
char *buffer;
send(OK);
```

3 Multithreading

To multithread, we have to create numerous worker threads that will carry out to process incoming requests distributed by the main thread, which is our main function. However, there must be some mutex and semaphores to prevent race conditions or busy waiting. I created a thread_mutex and a thread_in_use variables for those mechanisms.

Algorithm 22: Mutex and Semaphores

```
pthread_mutex_t thread_mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t thread_sem;
size_t thread_in_use = 0;
```

Since our main method is our dispatcher thread, we don't have to create a separate thread just for dispatch. Our dispatch will have to parse the incoming requests and sending them to a queue first. This is to handle the case where there are more requests than the amount of threads. To do this, we have to create a queue struct that is implemented in linked list. The linked list should be able to enqueue a node containing the request and dequeue a node containing the request. Each worker thread will get its own queue so there is less race conditions to worry about. I got the Linked List queue idea from GeeksforGeeks

Algorithm 23: Mutex and Semaphores

```
typedef struct QNode { ;
int key;
struct QNode∗ next;
}QNode;
typedef struct Queue { ;
struct QNode ∗front, ∗rear;
}Queue;
```

Algorithm 24: Enqueue

```
QNode* temp = newNode(k);
/* If queue is empty, then new node is front and rear both
   */
if (q->rear == NULL) { ;
q->front = q->rear = temp;
return;
} /* Add the new node at the end of queue and change rear
   */
q->rear->next = temp;
q->rear = temp;
```

Algorithm 25: Dequeue

```
/* If queue is empty, return NULL.                          */
if (q->front == NULL) return;
/* Store previous front and move front one node ahead      */
struct QNode* temp = q->front;
q->front = q->front->next; /* If front becomes NULL, then
   change rear also as NULL                                 */
if (q->front == NULL) q->rear = NULL;
return temp;
```

4 Logging and Healthcheck

Our code must handle concurrent logging from multiple threads. We cannot lock the file from other threads while one thread is writing to the log. What we can lock is the global offset calculation each thread has access to. Before each thread logs anything, they have to have access to the offset variable first. Once they have the variable, they lock it so other threads can't change its value until it is unlocked. Perform the calculation to find out how much space on the log this current thread need for this request. Then unlock the offset and proceed to log the request. The logging will be performed right before our server sends a response back.
Healthcheck is heavily incorporated in logging. We need to keep track of how many errors logged out of total entries logged. To do this, I increment a counter

for error entries and total entries every time a logging attempt is about to happen. Since the entries counts are shared between threads, it must be locked to perform changes to prevent race conditions. Only GET request is allow to access healthcheck. If GET requested a healthcheck without enabling logging, then we respond with 404 not found since the logging file doesn't exist to perform a healthcheck. If any other request other than GET is trying to access healthcheck, we respond with 403 forbidden as it is not allowed for other requests.

Algorithm 26: Logging

```
pthread_mutex_lock(log_mutex); p_offset = offset;
offset += strlen(fail_get) + strlen(raw_file_name) +
 strlen(error_message) + strlen(end_log);
++num_entries;
++err_entries;
pthread_mutex_unlock(log_mutex);
pwrite(log_file, log_entries, strlen(log_entries), p_offset);
p_offset += strlen(log_entries);
```

Algorithm 27: Healthcheck

```
if GET filename is healthcheck then
    if Logging enabled then
        send back err_entries
        nnum_entries
    else
        send 404 not found
else
    send 403 forbidden
```