EX3 – HTTP server

Goals:

In this programming assignment, you will write an HTTP server. Students are not required to implement the full HTTP specification, but only a very limited subset of it.
You will implement the following HTTP server that:

- Constructs an HTTP response based on the client's request.
- Sends the response to the client.

Program Description and What You Need to Do:

You will write two source files, server.c and threadpool.c.
The server should handle the connections with the clients. As we saw in class, when using TCP, a server creates a socket for each client it talks to. In other words, there is always one socket where the server listens to connections and for each client connection request, the server opens another socket. In order to enable a multithreaded program, the server should create threads that handle the connections with the clients. Since the server should maintain a limited number of threads, it constructs a thread pool. In other words, the server creates the pool of threads in advance, and each time it needs a thread to handle a client connection, it enqueues the request so an available thread in the pool will handle it.

Command line usage: server <port> <pool-size> <max-number-of-request>
Port is the port number your server will listen on, pool-size is the number of threads in the pool, and -number-of-request is the maximum number of requests your server will handle before it destroys the pool.

The server

In your program you need to:

1. Read request from the socket
2. Check input: The request's first line should contain a method, path, and protocol. Here, you only have to check that there are 3 tokens and that the last one is one of the HTTP versions, other checks on the method and the path will be checked later. In case the request is wrong, send a "400 Bad Request" response, as in file 400.txt.
3. You should support only the GET method, if you get another method, return the error message "501 not supported", as in file 501.txt
4. If the requested path does not exist, return the error message "404 Not Found", as in file 404.txt. The requested path is absolute, i.e. you should look for the path from the server root directory.

5. If the path is a directory but it does not end with a '/', return a “302 Found” response, as in 302.txt. Note that the location header should contain the original path + '/'. A real browser will automatically look for the new path.
6. If the path is a directory and it ends with a '/', search for index.html
    a. If index.html exists, return it.
    b. Otherwise, return the contents of the directory in the format as in the file dir_content.txt.
7. If the path is a file
    a. if the caller has no 'read' permissions, send a “403 Forbidden” response, as in file 403.txt. The file must have read permission for **everyone** and if the file is in some directory, all the directories in the path must have executing permissions.
    b. otherwise, return the file, format in file file.txt

When you create a response, you should construct it as follow:
The first line (version, status, phrase)\r\n
Server: webserver/1.0\r\n
Date: <date>\r\n (more later)
Location: <path>\r\n (only if the status is 302, otherwise omit this header)
Content-Type: <type/subtype>\r\n (more later)
Content-Length: <content-length>\r\n
Last-Modified: <last-modification-data>\r\n (more later)
Connection: close\r\n
\r\n

Response in Details:
First line example: HTTP/1.0 200 OK\r\n
The protocol is always HTTP/1.0.
The header "server" contains your server’s name, and it should be webserver/1.0.
In order to construct the date header, you can use:

```
#define RFC1123FMT "%a, %d %b %Y %H:%M:%S GMT"
time_t now;
char timebuf[128];
now = time(NULL);
strftime(timebuf, sizeof(timebuf), RFC1123FMT, gmtime(&now));
```

```
//timebuf holds the correct format of the current time.
```

Location is the new location in the case of header 302. In other words, the requested path + "/".

The content type is the mime type of the response body. You can use the following function:

```
char *get_mime_type(char *name)
{
        char *ext = strrchr(name, '.');
        if (!ext) return NULL;
        if (strcmp(ext, ".html") == 0 || strcmp(ext, ".htm") == 0) return "text/html";
        if (strcmp(ext, ".jpg") == 0 || strcmp(ext, ".jpeg") == 0) return "image/jpeg";
        if (strcmp(ext, ".gif") == 0) return "image/gif";
        if (strcmp(ext, ".png") == 0) return "image/png";
        if (strcmp(ext, ".css") == 0) return "text/css";
        if (strcmp(ext, ".au") == 0) return "audio/basic";
        if (strcmp(ext, ".wav") == 0) return "audio/wav";
        if (strcmp(ext, ".avi") == 0) return "video/x-msvideo";
        if (strcmp(ext, ".mpeg") == 0 || strcmp(ext, ".mpg") == 0) return "video/mpeg";
        if (strcmp(ext, ".mp3") == 0) return "audio/mpeg";
        return NULL;
}
```

If the function returns NULL, omit the content type header.

Content length is the length of the response body in bytes.

The last modification date is added only when the body is a file or a content of a directory. You should use the same format as in the header date.

Few comments:

1. Our server closes the connection after sending the response.
2. Don't use files to send error responses, this is very un-efficient.
3. When you fill your sockaddr_in struct, you can use htonl(INADDR_ANY) when assigning sin_addr.s_addr, meaning that the server listens to requests in any of its addresses.

4. When the server reads the request from the socket, it should read until there is "\r\n" in the read bytes. I.e. the server should read only the first line of the request.

The threadpool

The pool is implemented by a queue. When the server gets a connection (getting back from accept()), it should put the connection in the queue. When there will be available thread (can be immediate), it will handle this connection (read request and write response).
You should implement the functions in threadpool.h.
The server should first init the thread pool by calling the function create_threadpool(int). This function gets the size of the pool.

create_threadpool should:

1. Check the legacy of the parameter.
2. Create a threadpool structure and initialize it:
    a. num_thread = given parameter
    b. qsize=0
    c. threads = pointer to <num_thread> threads
    d. qhead = qtail = NULL
    e. Init lock and condition variables.
    f. shutdown = dont_accept = 0
    g. Create the threads with *do_work* as the execution function and the *pool* as an argument.

do_work should run in an endless loop and:

1. If the destruction process has begun, exit the thread
2. If the queue is empty, wait (no job to make)
3. Check again destruction flag.
4. Take the first element from the queue (*work_t)
5. If the queue becomes empty and the destruction process wait to begin, signal the destruction process.
6. Call the thread routine.

dispatch gets the pool, a pointer to the thread execution routine, and the argument to the thread execution routine. dispatch should:

1. Create work_t structure and init it with the routine and argument.
2. If destroy function has begun, don't accept new item to the queue

3. Add an item to the queue

destroy_threadpool

1. Set the don't_accept flag to 1
2. Wait for the queue to become empty
3. Set the shutdown flag to 1
4. Signal threads that wait on 'empty queue', so they can wake up, see the shutdown flag, and exit.
5. Join all threads
6. Free whatever you have to free.

Program flow:

1. Server creates a pool of threads; threads wait for jobs.
2. Server accepts a new connection from a client (aka a new socket fd)
3. Server dispatch a job - call dispatch with the main negotiation function and fd as a parameter (note that if you'll send a pointer to fd, the main will change the value of that pointer on the next accept). dispatch will add work_t item to the queue.
4. When there will be an available thread, it will take a job from the queue and run the negotiation function.

Error handling:

1. In any case of wrong command usage (you need to check that there are two positive numbers), print "Usage: server <port> <pool-size>\n"
2. In any case of a failure before connection with a client is set, use perror(<sys_call>) and exit the program.
3. In any case of a failure after connection with a client is set, if the error is due to some server-side error (like a failure in malloc), send "500 Internal Server Error", as in file 500.txt.

Useful function:

1. Strchr
2. Strstr
3. strtok
4. strcat

5. strftime
6. gmtime
7. scandir
8. opendir
9. readdir
10. stat
11. S_ISDIR
12. S_ISREG

Assumptions:

1. You can ignore headers on client requests. In other words, you need to parse only the first line.
2. You can assume that the length of the first line in the request is no more than 500 bytes.
3. When you're constructing a directory page, you can assume that the length of each entity line is no more than 500 bytes.

Compile the server:

Remember that you have to compile with the –lpthread flag.
Call your executable file 'server'.

What to submit:

You should submit a tar file called ex2.tar with server.c, threadpool.c and README. Find README instructions in the course web-site. **DON'T SUBMIT threadpool.h**

Test the server:

You can use a browser. The address line should be http://<computer-name>:<port-num>/<your-path>. If the server and browser are on the same machine, computer-name should be localhost.

Good-Luck!!