EX3 – HTTP server

Goals:

In this programming assignment you will write 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 A HTTP server that:

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

Program Description and What You Need to Do:

You will write two source files, server and threadpool.
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 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 construct a thread pool. In other words, the server create the pool of threads in advanced and each time it needs a thread to handle a client connection, it take one from the pool or enqueue the request if there is no available thread in the pool.

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

The server

In your program you need to:

1. Read request from socket
2. Check input: The request first line should contain 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 400 "Bad Request" respond, as in file 400.txt.
3. You should support only the GET method, if you get another method, return error message "501 not supported", as in file 501.txt

4. If the requested path does not exist, return error message "404 Not Found", as in file 404.txt. The requested path is relative to the server current directory, i.e. you should look for the path from the server root directory.
5. If path is directory but it does not end with a '/', return 302 Found response, as in 302.txt. Note that the location header contains the original path + '/'. Real browser will automatically look for the new path.
6. If path is directory and it ends with a '/', search for index.html
    a. If there is such file, return it.
    b. Else, return the contents of the directory in the format as in file dir_content.txt.
7. If the path is a file
    a. if the file is not regular (you can use S_ISREG) or the caller has no 'read' permissions, send 403 Forbidden response, as in file 403.txt. The file has to have read permission for everyone and if the file is in some directory, all the directories in the path have to have executing permissions for everyone.
    b. otherwise, return the file, format in file file.txt

When you create a response, you should construct it as follow:

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 status is 302)

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

Protocol is always HTTP/1.0.

The header "server" contains your server name and it should be webserver/1.0.

In order to construct the date, 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 case of header 302. In other words, the requested path + "/".

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 this function returns NULL, omit the header ‘content-type’.

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

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

<u>Few comments:</u>

1. Our server closes connection after sending the response.
2. Don't use the given 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 listen 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.

<u>The threadpool</u>

The pool is implemented by a queue. When the server gets a connection (getting back from accept()), it should create a work and put it in the queue. When there will be available thread (can be immediately), it will handle this work (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,int). This function gets the size of the pool and the max size of the queue.

create_threadpool should:

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

do_work should run in an endless loop and:

1. If destruction process has begun, exit 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 destruction process wait to begin, signal destruction process.
6. If destroy hasn't begun, signal dispatch that there is free space in the queue
7. Call the thread routine.

dispatch gets the pool, pointer to the thread execution routine and argument to 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. If queue is full, wait for free space
4. Add item to the queue
5. Signal that queue is not empty

destroy_threadpool

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

Program flow:

1. Server creates pool of threads, threads wait for jobs.
2. Server accept a new connection from a client (aka a new socket fd)
3. Server dispatch a job - call dispatch with the main negotiation function and its parameter. dispatch will add work_t item to the queue.
4. When there will be an available thread, it will takes a job from the queue and run the negotiation function.

Error handling:

1. In any case of wrong command usage, print "Usage: server <port> <pool-size> <max-queue-size> <max-number-of-request>\n"
2. In any case of a failure before connection with client is set, use perror(<sys_call>) and exit the program.
3. In any case of a failure after connection with client is set, if the error is due to some server side error (like 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 4000 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 ex3.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://localhost:<port-num>/<your-path>

Good-Luck!!