Networks HTTP Server Assignment 1560995 Documentation

Compiling and running the server

To compile the code, simply run the command:

```
$ make all
```

when in the `src/` directory. This will use the `Makefile` to compile the files and create an executable called `server` in the unzipped directory.

Before the server can be run, a certificate file and key file must be created for OpenSSL to use. To do this, run the command:

```
$ openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem
-days 1000
```

in the `src/` directory. This will create the files `cert.pem` and `key.pem`, using whatever password you set when running the command.

The server can then be run as follows (port number should be > 1024):

```
$ ./server <web-files> <port-number>
```

where the first argument is the directory containing the files to serve via the server, and the second is the port to bind to. For example, using the example `pages/` directory included in the zipfile on the port 5000 would be achieved by running:

```
$ ./server pages/ 5000
```

The server can then be connected to via a web browser, by visiting the url

```
https://localhost:<port-number>      (IPv4)
https://[::1]:<port-number>          (IPv6)
```

As this uses an SSL socket, the `https://` prefix **must** be used for the connection to be successful. The web browser will say this is an insecure connection due to the self-signed nature of the certificate, but this can be ignored and an exception added to test the server.

Structures

`typedef enum http_meth http_method`

Defines an HTTP header method enum to represent the HTTP methods GET, POST, and HEAD.

Values: `GET, HEAD, POST, UNKNOWN`

`typedef struct http_head http_header`

Defines an HTTP header structure to store a name/value pair found in an HTTP header, such as `Content-Length: 10`.

Contains: `char* name` - pointer to the name as a string
`char* value` - pointer to the value as a string

```
typedef struct http_req http_request
```

Defines an HTTP request structure to store a request to be processed.

Contains: `http_method method` - the HTTP method
`char* path` - pointer to the resource path as a string
`float version` - the HTTP version
`int header_num` - the number of header lines following
`http_header** headers` - the additional header lines
`int status` - the status of the request during processing
`int content_length` - the length of the content being served

Server Methods

```
void start_server(char* directory, char* port)
```

Begins the server, taking as arguments the directory where the resources are located and the port number to bind to (these are taken from the command line arguments). OpenSSL is initialized, and an SSL context created, using the certificate and private key files. The server information is placed into a `sockaddr_in6` structure and the socket created with `socket(AF_INET6, SOCK_STREAM, 0);` Once the socket is bound to the server address and port number, the `listen` and `accept` calls are made and the socket waits for a new client connection. When a client connects, a new pthread is created and calls the `handle_client` method with the new socket.

```
void* handle_client(void* param)
```

Called when a pthread is created, and is passed the socket through the param argument. The SSL structure is created and set to the new socket, then attempts to accept an SSL connection via the socket. If this is successful, a loop reads in 1 character at a time until it receives an entire HTTP request. As the request is read in, it is stored in a gradually increasing `char**`. The body is not required as the server is only serving required to serve content to the browser. Once the final `\r\n` of the header is read in, the request is parsed from the `char**` using the `parse_http` method, and then handled with the `handle_req` method, and then to clear up memory the `free_http` method is used. The `free_lines` method is also used to clear up the `char**`. Finally, the SSL socket is freed and the socket is closed, and the pthread detaches and exits.

```
http_request* parse_http(char** lines, int line_num)
```

Takes an HTTP request as a `char**` (each `char*` is a line of the request) and the total number of lines. This method then attempts to parse the lines into an HTTP request structure. The initial line is split into the method, resource path and HTTP version and added to the structure. The remaining lines are parsed into HTTP header structures and added to the request structure `http_header** headers` field. The content length is set to 0 initially. This method returns a pointer to the created HTTP request struct.

`void handle_req(SSL* ssl, http_request* req)`

Takes an HTTP request struct and the SSL struct as arguments. This method attempts to get the file requested using `get_file`, then sets the response header using `get_status` and `get_file_type` methods, then writes both the response header and response body (the file requested) to the SSL socket. The used memory is then free'd.

`char* get_file(http_request* req)`

Takes an HTTP request struct as an argument. This method attempts to open the file and read the contents into a `char*` buffer. If this works, the buffer is returned and the file is closed. If the file cannot be found, the request status is set to 404. If the path is forbidden, the status is set to 403. If the requested path is a directory, it attempts to return an `index.html` file (if one exists).

`char* get_file_type(http_request* req)`

Takes an HTTP request struct as an argument. Returns the type of the file pointed to by the request resource path, as a string to be used in the `Content-Type` header.

`char* get_status(int status)`

Takes a status code as an argument, and returns a string description. For example, the argument 200 will return `"OK"` and the argument 404 returns `"Not Found"`.

`char* append_dir(char* path, bool free_path)`

Takes a path and adds the directory (passed in by the command line arguments) to the front of it, then returns the result as a `char*`. If `free_path` is true, the memory passed as an argument has been alloc'd and is therefore free'd.

`void free_lines(char** lines, int line_num)`

Takes a `char**` and the number of `char*`s that have been alloc'd inside it, and frees them.

`void free_http(http_request* req)`

Takes an HTTP request struct and frees any alloc'd values inside it, and then frees the struct.

`void print_lines(char** lines, int line_num)`

Takes a `char**` and the number of `char*`s inside it, and prints a human readable version to the command line.

`void print_http(http_request* req)`

Takes an HTTP request struct and prints a human readable version to the command line.

`char* method_str(http_method meth)`

Takes an HTTP method enum and returns a string representation of it as a `char*`.