OPERATING SYSTEMS EXERCISE 3

HTTP is a text-based request-response application layer protocol, which is used to transfer files between a client and a server. Typically a client sends a request for a file located on the server. The server then responds and transmits the requested file.

HTTP requests start with a line which contains the request method (for instance `GET`), the location of the desired resource and the protocol version. This line is followed by an arbitrary number of header fields. Each header field is a line with a keyword and a value, separated by a colon (`:`). The request header ends with an empty line, which may be followed by a request body if the client wishes to transmit a resource to the server.

```
GET /index.html HTTP/1.1
Host: www.myhost.at
```

Example of a HTTP request: Request method `GET` is used to request the file `index.html` using HTTP version 1.1. The request header contains only one header field, the `Host` field, which specifies the hostname of the server. An empty line ends the header and since there is no body this is also the end of the request message.

The response starts with a line containing the protocol version, a status code and a textual description of the status code. As in the request, this line is followed by an arbitrary number of header fields and again the header ends with an empty line. The header is followed by the response body, which is typically the content of the requested file.

```
HTTP/1.1 200 OK
Date: Sun, 11 Nov 18 22:55:00 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 146
Last-Modified: Mon, 01 Oct 18 12:15:00 GMT
Connection: close

<!DOCTYPE html>
<html>
  <head>
    <title>My website</title>
  </head>
  <body>
    <h1>Hi!</h1>
    <p>This is my website.</p>
  </body>
</html>
```

Example of a HTTP response: The server responds to the above request with status code 200, which indicates that the request was successful. The header contains several fields with information about the content, such as the file format, the file length or the last time it was modified. The header ends with an empty line, which is followed by the file content.

The HTTP specification requires that each line ending in the header consists of two characters, a carriage return followed by a linefeed character, instead of the linefeed only which is standard on UNIX systems. In a C-string this sequence can be encoded as `"\r\n"`. Note that this does not apply to the content in the response body, which is transmitted as it is read from the file without modifications.

Assignment A – HTTP Client (10)

Write a client program which partially implement version 1.1 of the HTTP. The client takes an URL as input, connects to the corresponding server and requests the file specified in the URL. The transmitted content of that file is written to `stdout` or to a file.

```
SYNOPSIS
    client [-p PORT] [ -o FILE | -d DIR ] URL
EXAMPLE
    client http://www.nonhttps.com/
```

The client may be called with a number of options:

- Option `-p` can be used to specify the port on which the client shall attempt to connect to the server. If this option is not used the port defaults to 80, which is the standard port for HTTP.
- Either option `-o` or `-d` can be used to write the requested content to a file. Option `-o` is used to specify a file to which the transmitted content is written. Option `-d` is used to specify a directory in which a file of the same name (without the directory part, i.e. only the part after the last `/`) as the requested file is created and filled with the transmitted content. If the URL ends with `/` (thus requesting a directory) the filename is assumed to be `index.html`. If none of these options is given, the transmitted content is written to `stdout`.

The client creates a TCP/IP socket and connects to the hostname specified in the URL. For the purpose of this exercise we only consider URLs which start with the scheme identifier `http://`, immediatly followed by a hostname. For simplicity you may therefore assume that the hostname is the part after the initial `http://` and before any of the following characters:

`; / ? : @ = &`

Once the connection is established, the client sends a request for the file specified in the URL (the filepath is the part starting at and including the first `/` after the initial `http://`) using the HTTP method `GET`. The request header must start with the request line and must contain the `Host` field, which gives the hostname as specified in the URL. Also, the header field `Connection` with the value `close` should be added to the header (which tells the server that the connection should be closed after transmitting one file). For instance, the request header for the URL `http://www.nonhttps.com/` is:

```
GET / HTTP/1.1
Host: www.nonhttps.com
Connection: close
```

After transmitting its request, the client waits for a response from the server. Your client must correctly parse the status code in the first line of the response.

If the response header is invalid, the client prints the message `"Protocol error!"` and terminates with exit code 2. You can consider the response header to be invalid if the first line does not start with `HTTP/1.1` or if this is not followed by a number which can be parsed with `strtol(3)`.

If the response status in the first line of the response header is not 200, the client prints a message with the response status (status code and the textual description after it) and terminates with exit code 3.

The client may ignore any header fields and directly skip ahead to the content in the response body by looking for the empty line which ends the header. The remainder of the response is the content of the requested file and is eiter written to `stdout` or to a file, according to the options listed above. The client must not rely on the response header field `Content-Length` in order to determine the amount of data transfered, but instead read from the socket until the server closes the connection.

Assignment B – HTTP Server (10)

Write a server program which partially implement version 1.1 of the HTTP. The server waits for connections from clients and transmits the requested files.

```
SYNOPSIS
    server [-p PORT] [-i INDEX] DOC_ROOT
EXAMPLE
    server -p 1280 -i index.html ~/Documents/my_website/
```

The server may be called with following options:

- Option `-p` can be used to specify the port on which the server shall listen for incoming connections. If this option is not used the port defaults to 8080 (port 80 requires root privileges).
- Option `-i` is used to specify the index filename, i.e. the file which the server shall attempt to transmit if the request path is a directory. The default index filename is `index.html`.

The argument `DOC_ROOT` is the path of the document root directory, which contains the files that can be requested from the server.

The server listens for connections on the specified port and upon accepting a connection from a client, it reads the request message from the connection socket. Your server must be able to correctly identify the request method and the path of the requested resource.

If the request header is invalid, the server sends a response header with status code 400 (`Bad Request`) and closes the connection. You can consider the request header to be invalid if the first line does not consist of 3 values separated by whitespaces: first the request method, second the path of the requested file and third the protocol and version identifier `HTTP/1.1`.

The server must support the request method `GET`. If a different request method is sent by a client, the server sends a response header with status code 501 (`Not implemented`) and closes the connection.

If the first line of the request message is valid, the server may ignore all the rest of the request message and directly proceed to transmitting the requested file. The server prepends the document root path `DOC_ROOT` to the requested path[1]. If the path of the requested resource ends with a `/`, the default index filename (see options above) is appended to it. If the server cannot open the resulting filepath, it sends a response header with status code 404 (`Not Found`) and closes the connection.

If the file was successfully opened, the server first sends a response header with status code 200 (`OK`) and at minimum the fields `Date`, which contains the current date and time expressed in UTC (often also referred to as GMT) and encoded as specified in RFC 822[2], `Content-Length`, which gives the size of the requested file in bytes, and `Connection`, which should contain the value `close`. Your server may for instance transmit a response header as follows:

```
HTTP/1.1 200 OK
Date: Sun, 11 Nov 18 22:55:00 GMT
Content-Length: 146
Connection: close
```

Once the header is transmitted, the server proceeds with the transmission of the requested file, by reading the content of the file and writing it to the connection as is without any modifications. When the server is done with the transmission it closes the connection and waits for the next connection.

[1] For the purpose of this exercise you may simply concatenate the root path and the request path, for instance by using `strncat(3)`. Your server is not required to check whether the resulting path effectively lies within the document directory.

[2] RFC 822, Section 5: Date and Time Specification, `https://tools.ietf.org/html/rfc822#section-5`

The response header transmitted in case of an invalid request, unsupported request method or missing file must contain the header field `Connection` with the value `close`, but all other header fields can be omitted and no content is transmitted.

The server is not required to accept more than one connection at once. Therefore, you may simply accept the next connection after the previous connection has been closed.

The server must handle the signals `SIGINT` and `SIGTERM`. If any of these signals is received, the server completes an ongoing request and then terminates with exit code 0. If there are no open connections to clients when receiving the signal, the server terminates immediately with exit code 0.

Hints (for both assignments)

- When parsing the URL in order to extract the hostname and the filepath, the functions `strchr(3)`, `strpbrk(3)` or `strsep(3)` will be helpful to locate the first occurence of a character in a string.
- Since HTTP is a text-based protocol, you might want to use the functions of the stream API. This has the advantage that you can process the message headers line by line by using for instance `fgets(3)` or `getline(3)`.
- The first line of both the request and the response headers of the HTTP consists of 3 values separated by whitespaces. You might find it helpful to separate these using `strtok(3)`.
- You might want to use `strftime(3)` to format date and time as required for the response header. Take a close look at the 'Example' section of the man page for advice on date formatting in compliance with various standards.
- Be aware that the server must the read the entire request from the connection socket before writing anything to that socket. This is also true if your server is responding with an error status.

Testing

In order to test the functionality and correct implementation of the protocol, you should test your programs in following ways:

Assignment A – Your client requests files from servers on the internet

Using a web browser (e.g. Firefox), download a file which is publicly available on the internet, such as for instance `http://www.nonhttps.com/` and save it locally (e.g. under `~/Downloads/test.html`). Instead of a browser you can also use the command-line utility `wget`:

```
$ wget http://www.nonhttps.com/
```

Request the same file using your client and verify that the content of both files is identical using the utility `diff`.

```
$ ./client http://www.nonhttps.com/ > index.html
$ diff ~/Downloads/test.html index.html
$ mkdir dir
$ ./client -d dir http://www.nonhttps.com/index.html
$ diff ~/Downloads/test.html dir/index.html
```

Most websites recently refuse to transmit files via HTTP and instead require HTTPS, the encrypted version of HTTP. However, websites which still accept HTTP are for instance `http://neverssl.com/` or `http://www.nonhttps.com/`.

Assignment B – Your server processes requests from a web browser

Start your server and request files from it using a web browser (for instance using Firefox, which is the default browser in the lab). Again make sure that this works for various filetypes.

```
$ ls ./my_website
index.html
$ ./server ./my_website
```

When you type `127.0.0.1:8080` in the address bar of the browser (`127.0.0.1` is the IP address for localhost and `:8080` is used to specify the port), then your browser should request and display the file `index.html`. A browser will also request any requisites of a HTML document, such as images or scripts.

If you are working remotely via SSH you can also use `wget` instead of a browser with GUI:

```
$ wget -p -nd http://127.0.0.1:8080/
```

Assignment A and B – Your client requests files from your server

If you have implemented both, the server and the client, you may also test both programs together. Create a directory and populate it with one or more example files. Verify that requested files are transmitted correctly using the utility `diff`.

```
$ ls ./my_website
test.html
$ ./server -i test.html ./my_website &
$ ./client -o received.html -p 8080 http://localhost/
$ diff ./my_website/test.html received.html
```

Note that your implementation is not required to be able to transmit binary files (such as images). This is instead a bonus task (see next page).

Bonus Points

Bonus points are only awarded if both assignments, A (the client) and B (the server), have been implemented, are FULLY FUNCTIONAL and COMPLY WITH ALL INSTRUCTIONS! When adding features for bonus tasks, check that the basic requirements are still fulfilled, otherwise you risk loosing more points than you might gain!

- **Header field `Content-Type` (2):** Add support for the header field `Content-Type` to your server. This field is used to inform the client about the MIME-type[3] of the transmitted content. Your server should recognize following file types, based on the file name extension:

File extension	MIME type
`.html`, `.htm`	`text/html`
`.css`	`text/css`
`.js`	`application/javascript`

  2 bonus point are awarded if your server's response header contains the field `Content-Type` with the corresponding MIME type value if the requested file has any of the above extensions. If the file extension is none of those listed in the table, the field `Content-Type` should not be used.

- **Binary data (3):**

  HTTP is a text-based protocol, but it can also transmit binary data. If so far your client is reading the message body received from the server with `fgets(3)` or `getline(3)` and your server writes the message body with functions such as `fputs(3)`, then replace these with functions such as `fread(3)` and `fwrite(3)`, which are able to handle text and binary content alike. You should keep line-based functions for reading and writing the headers though.

  3 bonus points are awarded if your server is able to serve and your client is able to receive binary files, such as images.

- **Compression (5):** Implement compression of the transferred content in the gzip format. Make use of the zlib compression library[4] for compressing and decompressing the data. You can use it by including the `zlib.h` header:

  ```
  #include <zlib.h>
  ```

  Also, you will have to link the zlib library by adding `-lz` to your linker options. Make sure to test this in the lab in order to avoid linker warnings or errors.

  5 bonus points are awarded if your server is able to compress data into the gzip format and your client is able to decompress such data.

  Your server must recognize clients which accept data encoded in gzip format (by searching for the field `Accept-Encoding` in the request header and checking whether it contains the value `gzip`), and respond to requests from these clients with the line

  ```
  Content-Encoding: gzip
  ```

  in the response header and correctly compress the content in the body into that format. Test your server by requesting files from your server using a web browser which supports gzip, for instance Firefox!

  Your client must advertise his decompression capability with the line

  ```
  Accept-Encoding: gzip
  ```

  in the request header and be able to correctly decompress content sent with that encoding. Test your client by requesting files from a web server which supports gzip, such as `http://www.nonhttps.com/`.

[3] Media type, a two-part identifier for file formats, see `https://en.wikipedia.org/wiki/Media_type`
[4] `http://zlib.net/`

Coding Rules and Guidelines

Your score depends upon the compliance of your submission to the presented guidelines and rules. Violations result in deductions of points. Hence, before submitting your solution, go through the following list and check if your program complies.

Rules

Compliance with these rules is essential to get any points for your submission. A violation of any of the following rules results in 0 points for your submission.

1. All source files of your program(s) must compile via

   ```
   $ gcc -std=c99 -pedantic -Wall -D_DEFAULT_SOURCE -D_BSD_SOURCE -D_SVID_SOURCE
                                  -D_POSIX_C_SOURCE=200809L -g -c filename.c
   ```

   without *errors* and your program(s) must link without *errors*. The compilation flags must be used in the Makefile. The feature test macros must not be bypassed (i.e., by undefining these macros or adding some in the C source code).

2. The functionality of the program(s) must conform to the assignment. The program(s) shall operate according to the specification/assignment given the test cases in the respective assignment.

General Guidelines

Violation of following guidelines leads to a deduction of points.

1. All source files of your program(s) must compile with

   ```
   $ gcc -std=c99 -pedantic -Wall -D_DEFAULT_SOURCE -D_BSD_SOURCE -D_SVID_SOURCE
                                  -D_POSIX_C_SOURCE=200809L -g -c filename.c
   ```

   without *warnings and info messages* and your program(s) must link without warnings.

2. There must be a Makefile implementing the targets: `all` to build the program(s) (i.e. generate executables) from the sources (this must be the first target in the Makefile); `clean` to delete all files that can be built from your sources with the Makefile.

3. The program shall operate according to the specification/assignment without major issues (e.g., segmentation fault, memory corruption).

4. Arguments have to be parsed according to UNIX conventions (we strongly encourage the use of `getopt(3)`). The program has to conform to the given synopsis/usage in the assignment. If the synopsis is violated (e.g., unspecified options or too many arguments), the program has to terminate with the usage message containing the program name and the correct calling syntax. Argument handling should also be implemented for programs without arguments.

5. Correct (=normal) termination, including a cleanup of resources.

6. Upon success the program has to terminate with exit code 0, in case of errors with an exit code greater than 0. We recommend to use the macros `EXIT_SUCCESS` and `EXIT_FAILURE` (defined in `stdlib.h`) to enable portability of the program.

7. If a function indicates an error with its return value, it *should* be checked in general. If the subsequent code depends on the successful execution of a function (e.g. resource allocation), then the return value *must* be checked.

8. Functions that do not take any parameters have to be declared with `void` in the signature, e.g., int get_random_int(void);.

9. Procedures (i.e., functions that do not return a value) have to be declared as `void`.

10. Error messages shall be written to `stderr` and should contain the program name `argv[0]`.

11. It is forbidden to use the functions: `gets`, `scanf`, `fscanf`, `atoi` and `atol` to avoid crashes due to invalid inputs.

FORBIDDEN	USE INSTEAD
gets	fgets
scanf	fgets, sscanf
fscanf	fgets, sscanf
atoi	strtol
atol	strtol

12. Documenation is mandatory. Format the documentation in Doxygen style (see Wiki and Doxygen's intro).

13. Write meaningful comments. For example, meaningful comments describe the algorithm, or why a particular solution has been chosen, if there seems to be an easier solution at a first glance. Avoid comments that just repeat the code itself
(e.g., i = i + 1; /* i is incremented by one */).

14. The documentation of a module must include: name of the module, name and student id of the author (`@author` tag), purpose of the module (`@brief`, `@details` tags) and creation date of the module (`@date` tag).

    Also the Makefile has to include a header, with author and program name at least.

15. Each function shall be documented either before the declaration or the implementation. It should include purpose (`@brief`, `@details` tags), description of parameters and return value (`@param`, `@return` tags) and description of global variables the function uses (`@details` tag).

    You should also document `static` functions (see `EXTRACT_STATIC` in the file `Doxyfile`). Document visible/exported functions in the header file and local (`static`) functions in the C file. Document variables, constants and types (especially `structs`) too.

16. Documentation, names of variables and constants shall be in English.

17. Internal functions shall be marked with the `static` qualifier and are not allowed to be exported (e.g., in a header file). Only functions that are used by other modules shall be declared in the header file.

18. All exercises shall be solved with functions of the C standard library. If a required function is not available in the standard library, you can use other (external) functions too. Avoid reinventing the wheel (e.g., re-implementation of `strcmp`).

19. Name of constants shall be written in upper case, names of variables in lower case (maybe with fist letter capital).

20. Use meaningful variable and constant names (e.g., also semaphores and shared memories).

21. Avoid using global variables as far as possible.

22. All boundaries shall be defined as constants (macros). Avoid arbitrary boundaries. If boundaries are necessary, treat its crossing.

23. Avoid side effects with `&&` and `||`, e.g., write if (b != 0) c = a/b; instead of if (b != 0 && c = a/b).

24. Each `switch` block must contain a `default` case. If the case is not reachable, write `assert(0)` to this case (defensive programming).

25. Logical values shall be treated with logical operators, numerical values with arithmetic operators (e.g., test 2 strings for equality by strcmp (...) == 0 instead of !strcmp (...)).

26. Indent your source code consistently (there are tools for that purpose, e.g., `indent`).

27. Avoid tricky arithmetic statements. Programs are written once, but read more times. Your program is not better if it is shorter!

28. For all I/O operations (read/write from/to `stdin`, `stdout`, files, sockets, pipes, etc.) use *either* standard I/O functions (`fdopen(3)`, `fopen(3)`, `fgets(3)`, etc.) *or* POSIX functions (`open(2)`, `read(2)`, `write(2)`, etc.). Remember, standard I/O functions are buffered. Mixing standard I/O functions and POSIX functions to access a common file descriptor can lead to undefined behaviour and is therefore forbidden.

29. If asked in the assignment, you must implement signal handling (`SIGINT`, `SIGTERM`). You must only use *async-signal-safe* functions in your signal handlers.

30. Close files, free dynamically allocated memory, and remove resources after usage.

31. Don't waste resources due to inconvenient programming. Header files shall not include implementation parts (exception: macros).

Exercise 3 Guidelines

Violation of following guidelines leads to a deduction of points in exercise 3.

1. It is forbidden to use the deprecated functions `gethostbyname`, `gethostbyaddr`, `gethostbyname2`, `gethostbyname_r`, `gethostbyname2_r` or any other function of that family documented on the man page `gethostbyname(3)`. Use `getaddrinfo(3)` instead.