Practical 1: Non-blocking Web server

Due Mar 27 by 17:00 **Points** 35 **Available** until Jun 21 at 23:59

Please Note: Before attempting this practical please study the material on **Sockets. pdf** and the tutorials under Week 2 lecture materials I will not be covering socket programming in the lecture and this is left for you as a learning exercise and part of practical 3. If you are having a lot of issues with coding, please see the helpful staff at the Computer Science Learning Center (https://ecms.adelaide.edu.au/computer-science/computer-science-learning-centre).

Learning Outcomes

1. Reading a Protocol Specification

Most of the application layer protocols are relatively simple and text-based. This exercise gets you to first read the protocol specification(s). The RFC documents are not very typical of a standards document - they are very compact. They are also somewhat incomplete - they come with a standard reference implementation which is used to resolve ambiguities in the specification. So the first learning outcome is the ability to read a protocol specification.

2. Practical Experience with Socket API

The second outcome is some practical experience with using the socket API. This is quite straightforward in reality, but there are quite a few "not immediately obvious" tricks.

3. Understanding the HTTP

The third learning outcome is a detailed knowledge of the protocol itself. HTTP is simple and not dissimilar to many other application layer protocols. Once you see how this works, it becomes much easier to implement other protocols (Simple Mail Transfer Protocol, SMTP, for example) and to add functionality to existing applications ("send this information to a friend")

Note that the focus is *not* the programming but the protocol. There are generally libraries available in high-level languages that support HTTP and other protocols. If you were building a service you would most likely use these libraries rather than working at the socket layer. However, those libraries hide much of the protocol behavior we are trying to study, which is our motivation for working at the socket layer.

Acknowledgement: This assignment is Adapted from Kurose & Ross - Computer Networking: a top-down approach featuring the Internet. It is *not* the same though so make sure you are following the specifications and using the provided code from this practical sheet and not the textbook or website.

CBOK categories: Abstraction, Design, Data & Information, Networking, and Programming.

Submitting your assignment

You will need to keep your files in SVN. If you aren't familiar with SVN, the following SVN notes will assist you.

The handin key for this practical is `<year>/<semester>/cna/webserver`

You can submit and automatically mark your assignment by following **this link** (https://cs.adelaide.edu.au/services/websubmission/) to the web-submission system. Note that this is just a preliminary test to make sure your Web server meets the basic functionality. It's your responsibility to ensure it meets the specifications for the methods your are asked to implement in this practical.

Template files

This practical comes with a skeleton web server and helper methods to reduce complexity. Download and extract the following zip file into your repository.

prac1_files.zip

We will go through these files in the next section.

The practical is set up so that all code you need will belong inside the comment tags below, where X ranges from 1 to 11.

```
/* START CODE SNIPPET X */
...
/* END CODE SNIPPET X */
```

Introduction

In this practical, we will develop a web server. In the end, you will have built a non-blocking web server that is capable of processing multiple simultaneous service requests in parallel. You should be able to demonstrate that your Web server is capable of delivering your home page to a web browser and that your server returns standards compliant responses.

Resources to study

To understand the data that is sent to and from a web server you will need to study the HTTP. Use the following resources and sections to help you understand the key data points that need to be sent and received.

- **RFC 1945** (https://www.cs.adelaide.edu.au/users/third/cn/RFCs/rfc1945.txt)
- **Wireshark HTTP lab quiz**
- Verbose cUrl command

For the purposes of this practice we will only be looking at HTTP/1.1 and older requests. HTTPS requests are out of the scope of this practical.

RFC 1945

Most of the application layer protocols are relatively simple and text-based. The RFC documents are not very typical of a standards document - they are very compact. They are also somewhat incomplete - they come with a standard reference implementation which is used to resolve ambiguities in the specification. It is important to be able to read a protocol specification.

For this practical, you will need to understand the lines (text) that gets sent to the server on an HTTP request, and (at the very least) the first line that is returned. Scan the contents page for the keywords; **request** and **response**.

Wireshark HTTP Lab Quiz

In doing the Wireshark HTTP lab, you would have seen the data that is sent to and from a web server. There is a lot of extraneous data that is sent which is beyond the scope of this practical.

Identify the key data bits of data.

Food for thought: What data would you change to cause the HTTP request to fail?

Verbose cUrl command

The cUrl command is a command line web request tool. By enabling the `-v` (verbose) option you can see the data sent to and from a web server.

The following command is to the URI http://jsonplaceholder.typicode.com/todos/1 (http://jsonplaceholder.typicode.com/todos/1)

```
 1  $ curl -v http://jsonplaceholder.typicode.com/todos/1
 2  *   Trying 104.24.106.213...
 3  * TCP_NODELAY set
 4  * Connected to jsonplaceholder.typicode.com (104.24.106.213) port 80 (#0)
 5  > GET /todos/1 HTTP/1.1
 6  > Host: jsonplaceholder.typicode.com
 7  > User-Agent: curl/7.54.0
 8  > Accept: */*
 9  >
10  < HTTP/1.1 200 OK
11  < Date: Mon, 25 Feb 2019 23:07:55 GMT
12  < Content-Type: application/json; charset=utf-8
13  < Content-Length: 83
14  < Connection: keep-alive
15  < Set-Cookie: __cfduid=d4765bb5a80e476f748c1625a4b41109d1551136075; expires=Tue, 25-Feb-20 23:07:55 GMT; path=/; domain=.typicode.com; HttpOnly
16  < X-Powered-By: Express
17  < Vary: Origin, Accept-Encoding
18  < Access-Control-Allow-Credentials: true
19  < Cache-Control: public, max-age=14400
20  < Pragma: no-cache
21  < Expires: Tue, 26 Feb 2019 03:07:55 GMT
22  < X-Content-Type-Options: nosniff
23  < Etag: W/"53-hfEnumeNh6YirfjyjaujcOPPT+s"
24  < Via: 1.1 vegur
25  < CF-Cache-Status: HIT
26  < Accept-Ranges: bytes
27  < Server: cloudflare
28  < CF-RAY: 4aedd4b8b9ce1d44-MEL
29  <
30  {
31    "userId": 1,
32    "id": 1,
33    "title": "delectus aut autem",
34    "completed": false
35  * Connection #0 to host jsonplaceholder.typicode.com left intact
36  }
```

In the output above `>` denotes data sent to the server and `<` denotes data received from the server. Study lines **4**, **5**, **6** and **10** from the above carefully.

Compile and Run

To compile the Web server run:

```
$ gcc helpers.c web_server.c -o web_server
```

To run the Web server run:

```
$ ./web_server [port]
```

When you run the program for the first time it will terminate with no output. That is ok for now, we will fix them in as we go along.

In the following steps, we will go through the code for the first implementation of the Web server in C. Follow along with the `web_server.c` file.

Non-blocking

Our first implementation of the Web server will be non-blocking, where the processing of each incoming request will take place inside a separate process. This allows the server to service multiple clients in parallel, or to perform multiple file transfers to a single client in parallel.

In the code below, this is done with `fork()`. This creates a new process with a *copy* of the original process' variables. Threads can also be used to serve multiple requests in parallel. In this case, both threads *share* variables. This is more efficient as a copy doesn't have to be made, but it also requires care to make sure that both threads don't try to change values at the same time and that changes happen in the right order. If you are keen to use threads instead, you are welcome to change the code.

The basic structure of the code is:

```c
int main(int argc, char *argv[])
{
  while (true)
  {
    if ((pid = fork()) == 0)
    {
      /*----------START OF CHILD CODE----------------*/
      /* We are now in the child process */

      ...

      /* All done return to parent */
      exit(EXIT_SUCCESS);
    }
    /*----------END OF CHILD CODE----------------*/

    /* Back in parent process
     * if child exited, wait for resources to be released
     */
    waitpid(-1, NULL, WNOHANG);
  }
}
```

The parent code will listen for new connection requests and each time a new connection is received a child is created and will run code to handle the new connection: reading the HTTP request and sending an HTTP response on the connection.

The parent will continue separately and go back to the start of the loop accepting further connections.

Variables

The main program starts by defining variables that will be used in the code.

It is important to understand the purpose of each variable.

The server will have two sockets:

1. The `listen_socket` variable is for the socket that listens and accepts new requests

2. The `connection_socket` variable is to hold the connection to a client.

It is very important that you send and receive requests on the correct socket.

The `response_buffer` is just a variable to hold the response that your server will send back to the client. It is set to the maximum response size which can be changed in the `config.h` file.

`status_code` and `status_phrase` will be used to hold the HTTP response code and phrase that should be sent back to the client.

```c
int main(int argc, char *argv[])
{
  /* structure to hold server's and client addresses, respectively */
  struct sockaddr_in server_address, client_address;

  int listen_socket = -1;
  int connection_socket = -1;
  int port = 0;

  /* id of child process to handle request */
  pid_t pid = 0;

  char response_buffer[MAX_HTTP_RESPONSE_SIZE] = "";
  int status_code = -1;
  char *status_phrase = "";

  ...
}
```

Step 1: Create a socket

The first step for the server is to create a socket. You will need to write the code to create the socket. Be sure to use the correct variable (either `listen_socket` or `connection_socket`).

If you are not sure how to create a socket referer to:

- The lecture notes
- socket man page `$ man 2 socket`
- **Online socket tutorial** (https://www.binarytides.com/socket-programming-c-linux-tutorial/)

```c
/* 1) Create a socket */
/* START CODE SNIPPET 1 */
...
/* END CODE SNIPPET 1 */
```

Specifying the port

Normally, Web servers process service requests that they receive through the well-known port number 80. You can choose any port higher than 1024, but remember to direct any requests to your Web server with the corresponding port. The following code sets the port to either the value typed on the command line, or if no port is given, then the `DEFAULT_PORT`, which is defined in the `config.h` file.

```
/* Check command-line argument for port and extract
 * port number if one is specified. Otherwise, use default
 */
if (argc > 1)
{
  /* Convert from string to integer */
  port = atoi(argv[1]);
}
else
{
  port = DEFAULT_PORT;
}

if (port <= 0)
{
  /* Test for legal value */
  fprintf(stderr, "bad port number %d\n", port);
  exit(EXIT_FAILURE);
}
```

Steps 2 & 3: Binding

Next, you need to bind the socket you created to the port and network interfaces that it should listen to. You will need to write the code to set the correct values in the `server_address` structure and call `bind()` to bind the address information to the socket.

The `memset` function makes sure that `serv_addr` doesn't have any values in it (i.e. it will clear the data structure so that it contains all 0's).

See `$ man 2 bind` for how to use the `bind()` method

See **https://linux.die.net/man/7/ip** (https://linux.die.net/man/7/ip) on how to set the values in `serv_addr.`

```
/* 2) Set the values for the server address structure */
/* START CODE SNIPPET 2 */
...
/* END CODE SNIPPET 2 */

/* 3) Bind the socket to the address information set in server_address */
/* START CODE SNIPPET 3 */
...
/* END CODE SNIPPET 3 */
```

Steps 4 & 5: Listening

The server should now start listening for a TCP connection request on the socket. Once it is listening, it can begin accepting connections. Because we will be servicing request messages indefinitely, we place the accept request operation inside of an infinite loop. This means we will have to terminate the Web server by pressing `^C` (Ctrl-C) on the keyboard.

You need to add the code to make the socket listen and to accept a connection.

the accept method will block until a connection is made.

See `$ man 2 listen` and `$ man 2 accept` for how to use the `listen()` and `accept()` methods

```
/* 4) Start listening for connections */
/* START CODE SNIPPET 4 */
...
/* END CODE SNIPPET 4 */

/* Main server loop
 * Loop while the listen_socket is valid
 */
while (listen_socket >= 0)
{
  /* 5) Accept a connection */
  /* START CODE SNIPPET 5 */
  ...
  /* END CODE SNIPPET 5 */
```

Handling a connection

BEYONCE, CAN YOU HANDLE THIS?
I DON'T THINK YOU CAN HANDLE THIS

When a connection request is received, we create a child process to handle the request in a separate process and close any sockets the child is not using (remember the child gets a *copy* of the sockets. Closing the child socket does not close the parent's copy only it's own copy).

After the child has started execution, the main (parent) process closes its copy of the connected socket and returns to the top of the request processing loop to accept another connection. The main process will then block, waiting for another TCP connection request, while the child continues running. When another TCP connection request is received, the main process goes through the same process of child process creation regardless of whether the previous child has finished execution or is still running.

Note the `wait()` call below allows the system to free up resources from the child when a child exits. If no child has exited, `wait()` with `NOHANG` returns immediately.

```
/* Fork a child process to handle this request */
if ((pid = fork()) == 0)
{
  /*----------START OF CHILD CODE----------------*/
  /* We are now in the child process */

  /* Close the listening socket
    * The child process does not need access to listen_socket
    */
  if (close(listen_socket) < 0)
  {
    fprintf(stderr, "child couldn't close listen socket\n");
    exit(EXIT_FAILURE);
  }

  ...

  /* All done return to parent */
  exit(EXIT_SUCCESS);
}
/*----------END OF CHILD CODE----------------*/

...

/* if child exited, wait for resources to be released */
waitpid(-1, NULL, WNOHANG);

```

Step 6: Reading the request

Once we have closed the `listen_socket` in the child process, we create an `http_request` structure which will be used to store the information about the HTTP request, such as it's `method` and `URI` and pass this structure to the helper function `Parse_HTTP_Request()`. This helper will read the request from the given socket and fill in the `http_request` structure for you. The interface information about the helper functions is in the file `helpers.h` and the file `helpers.c` contains the implementation. ***Note that we have provided these to help you and are not supposed to be an exhaustive list of functions required to complete the practical. However, most of what you need is given to you in these helper files so the helper files tries to make this practical much easier for you by doing most of the hard work for you.***

```
/* See httpreq.h for definition */
struct http_request new_request;
/* 6) call helper function to read the request
 * this will fill in the struct new_request for you
 * see helper.h and httpreq.h
 */
/* START CODE SNIPPET 6 */
...
/* END CODE SNIPPET 6 */
```

`new_request` will now be filled with the method and URI that the client sent.

Closing the connection

Finally, the child is finished and can close any sockets it still has open and exit.

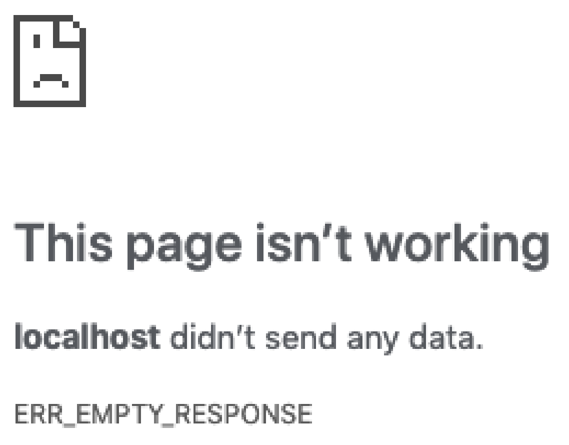
```c
/* Back in parent process
 * Close parent's reference to connection socket,
 * then back to top of loop waiting for next request
 */
if (connection_socket >= 0)
{
  if (close(connection_socket) < 0)
  {
    fprintf(stderr, "closing connected socket failed\n");
    exit(EXIT_FAILURE);
  }
}

/* if child exited, wait for resources to be released */
waitpid(-1, NULL, WNOHANG);
```

Now we are ready to connect to a browser.

Connecting to a browser

After your program successfully compiles, run it with an available port number (e.g. 8080, 9000, etc), and try contacting it from a browser.

Enter `http://localhost:[port]/index.html` into the address bar of your browser, replacing `[port]` with the port number for your Web server and press enter. The browser will display an error because our Web server has accepted the request, closed the connection and not sent anything back.


This page isn't working

localhost didn't send any data.

ERR_EMPTY_RESPONSE

In the terminal (that is running your Web server) the server should display the contents of the HTTP request message.

```
Waiting connection on port 8080...
received request: GET /index.html HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.109 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-AU,en;q=0.9,en-US;q=0.8


Method is: GET
URI is: /index.html
version is: HTTP/1.1
Sending response line:
```

Step 7: Parsing HTTP Request

Instead of simply displaying the browser's HTTP request message, we will analyze the request and send an appropriate response. We are going to ignore the information in the header lines, and use only the path contained in the request line.

You now need to write code that will decide, based on the results from `Parse_HTTP_Request()`, what the `status_code` and `status_phrase` variables should be set to for the response.

Your server is required to handle and return the correct status code and phrase for the following situations:

1. The requested resource is found
2. The requested resource is not found
3. The requested method isn't implemented (your server will only implement GET and HEAD methods)
4. The client sent an invalid request

Look in the [RFC 1945](https://www.cs.adelaide.edu.au/users/third/cn/RFCs/rfc1945.txt) (https://www.cs.adelaide.edu.au/users/third/cn/RFCs/rfc1945.txt) specification to work out the most suitable status code and phrase.

You can use the `-X [method]` option in cUrl to run web requests on your Web server.

```
$ curl -v -X GET localhost:8080/index.html
```

Fill in the code needed at:

```
/* 7) Decide which status_code and reason phrase to return to client */
/* START CODE SNIPPET 7 */
...
/* END CODE SNIPPET 7 */
```

Having determined the appropriate response, we can now send the response header. Check the HTTP response format to make sure you fill this in correctly!

Step 8: Formatting the response

```
/* 8) Set the reply message to the client
 * Copy the following line and fill in the ??
 * sprintf(response_buffer, "HTTP/1.0 %d %s\r\n", ??, ??);
 */
/* START CODE SNIPPET 8 */
sprintf(response_buffer, "HTTP/1.0 %d %s\r\n", ??, ??);
/* END CODE SNIPPET 8 */
```

Step 9: Sending the response

Now we can send the status line and our header lines to the browser by writing to the socket. Be sure you send to the correct socket.

```
printf("Sending response line: %s\n", response_buffer);

/* 9) Send the reply message to the client
 * Copy the following line and fill in the ??
 * send(??, response_buffer, strlen(response_buffer), 0);
 */
/* START CODE SNIPPET 9 */
send(??, response_buffer, strlen(response_buffer), 0);
/* END CODE SNIPPET 9 */
```

Steps 10 & 11: Preparing a HTTP Response

Now that the status line has been placed on the socket on its way to the browser, it is time to do the same with the response headers and the entity body. We need to decide whether or not an entity body should be returned.

You will need to understand the difference between GET and HEAD.

We can use the helper methods `Is_Valid_Resource()` to see if the requested file exists. If the requested file does exist, we can call helper function `Send_Resource()` to send the Content-Length header and the file contents on the socket (see `helpers.h` for how to use `Is_Valid_Resource()` and `Send_Resource()`).

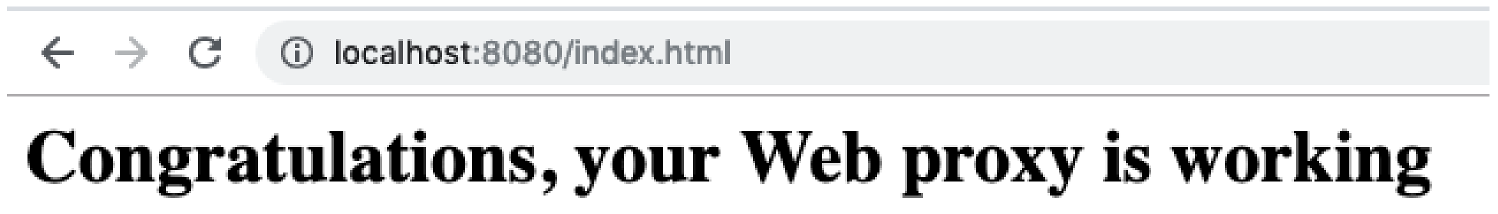
After sending the entity body, the work in the child has finished, so we close the socket before terminating.

```
bool is_ok_to_send_resource = false;
/* 10) Send resource (if requested) under what condition will the
 * server send an entity body?
 */
```

```
/* START CODE SNIPPET 10 */
is_ok_to_send_resource = ...
/* END CODE SNIPPET 10 */

if (is_ok_to_send_resource)
{
  Send_Resource(connection_socket, new_request.URI);
}
else
{
  /* 11) Do not send resource
   * End the HTTP headers
   * Copy the following line and fill in the ??
   * send(??, "\r\n\r\n", strlen("\r\n\r\n"), 0);
   */
  /* START CODE SNIPPET 11 */
  send(??, "\r\n\r\n", strlen("\r\n\r\n"), 0);
  /* END CODE SNIPPET 11 */
}
```

Running the Web server

When you have completed all the steps correctly your web browser will be able to respond with a message when running `http://localhost:[port]/index.html`


localhost:8080/index.html

Congratulations, your Web proxy is working

You should also make sure that your Web server will work with HEAD and other requests. You may use the cUrl command to test your web server.

Last Words

We have only coded the very basics of a web server. There is a lot of missing, for instance, we do not return the content type of the file, we don't return an entity body when there is an error (which means nothing will be displayed in the browser when an error occurs). If you have extra time, look at implementing some of these.

This completes the code for the second phase of development of your Web server. Try adding some web pages or text files to the `RESOURCE_PATH` directory (the `RESOURCE_PATH` directory is defined in `config.h`) Create this directory where you are running the server and try viewing your files with a browser. Remember to include a port specified in the URL of your home page, so that your browser doesn't try to connect to the default port 80.

When you connect to the running web server with the browser, examine the GET message requests that the web server receives from the browser. You can use a tool called telnet to check that your server responds correctly to HEAD requests.

Prac 1: Web Proxy (1)

Criteria	Ratings		Pts
Successful compilation of code based on the skeleton provided. If your server crashes for reasons that are not due to a failure of the websubmission system (for example submitting solutions that are not tested on the University Linus image or with inadequate amount of testing and with software bugs), we will perform no further manual inspections or testing and you will receive an automatic ZERO for the whole assignment.	**1.0 pts Full Marks**	**0.0 pts No Marks**	1.0 pts
Server started up successfully If your server crashes for reasons that are not due to a failure of the websubmission system (for example submitting solutions that are not tested on the University Linus image or with inadequate amount of testing and with software bugs), we will perform no further manual inspections or testing and you will receive an automatic ZERO for the whole assignment.	**4.0 pts Full Marks**	**0.0 pts No Marks**	4.0 pts
Server acceptance of client request for html file and correct response	**5.0 pts Full Marks**	**0.0 pts No Marks**	5.0 pts
Correct server behaviour for a request for non existent file	**5.0 pts Full Marks**	**0.0 pts No Marks**	5.0 pts
Correct server behaviour to a HEAD method	**5.0 pts Full Marks**	**0.0 pts No Marks**	5.0 pts
Correct server behaviour to an unimplemented method	**5.0 pts Full Marks**	**0.0 pts No Marks**	5.0 pts
Correct sever behaviour for a bad request	**5.0 pts Full Marks**	**0.0 pts No Marks**	5.0 pts
Server successfully handles multiple requests (non-blocking)	**5.0 pts Full Marks**	**0.0 pts No Marks**	5.0 pts
			Total Points: 35.0