CSCI3120
Assignment 1*

Instructor: Alex Brodsky

Due: 12:00pm (noon), Tuesday, May 24, 2016

The purpose of this assignment is to refresh your coding abilities in C, and to serve as an example of the difficulty, style, and format of assignments that you can expect in this course. In this assignment you will:

1. Implement a simple web server
2. Create a test suite for the simple web server, and
3. Answer a couple questions.

You will need to checkout the supporting code from the SVN repository `https://svn.cs.dal.ca/csci3120/<CSID>`, where `<CSID>` refers to your CS login. The supporting code contains the networking code needed to complete this assignment. Please follow the guidelines in Section 6.1.

1 Background

Web servers are almost as ubiquitous as operating systems today. Many systems, even consumer-grade ones, can and do run web servers, sometimes as part of the OS itself. Furthermore, web servers have to perform many of the same functions and decisions that operating systems do. Consequently, using web servers to study how operating systems work is not such an outlandish concept.

The function of a web server is to serve web requests from web clients (web browsers). In short, to view a web page or download content, a web client (browser) makes a network connection (over the Internet) to a web server. The client then sends a *request* to the server, specifying the file that the web server should send back. The web server receives and parses the request, checks if the file is available, and sends it back if it is. The web client then uses and renders the file as it deems fit.

When a web server starts up, it first binds to a *port*, which allows clients to unambiguously specify which server to connect to. For example, most web servers bind to port 80. However, since our servers will be run as user-level processes, only ports in the range 1024 to 65335 may be used. Once the server binds to a port and completes any additional initialization, it blocks (goes to sleep) and waits for clients to connect.

To connect to a server, the client uses two identifiers: the host name (or IP address) and the port number. For this assignment you will be running both the client and server on the same machine, so the host name will be *localhost* and the port will be whatever the server is bound to. When the client connects to the server, it blocks until the server is ready to serve the client. When

*Revised: May 2, 2016

the server is ready, it *accepts* the connection from the client. At this point a bidirectional pipe is created between the client and server, meaning that the client can send data to the server and the server can send data to the client. At this point the client can send its request to the server, using the Hypertext Transfer Protocol (HTTP).

An HTTP request message from the client to the server consists of four parts:

1. A request line
2. One or more request headers
3. Am empty line
4. An optional message body

For our simple web server, the request line will be of the form:

```
GET <File> HTTP/1.1
```

where `<File>` is a `/` followed by the path of the file. Our simple web server can safely ignore the rest of the request, including the request headers, the empty line, and the optional message body. For example, the request to Dal's web server for the CS web page looks like this:

```
GET /faculty/computerscience/programs.html HTTP/1.1
Host: www.dal.ca
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us
Connection: keep-alive
Accept-Encoding: gzip, deflate
```

Once the simple web server receives the request, it parses the first line to extract the path of the file being requested. It then opens and sends back a response message to the client, which includes the contents of the file. If the file does not exist, or the request is malformed, an error response is sent.

An HTTP response message from the server to the client consists of four parts:

1. A status line
2. Zero or more response headers
3. Am empty line
4. An optional message body containing the requested file.

For our simple web server, the status line will take one of three forms:

`HTTP/1.1 400 Bad request`
: If the request message is malformed

`HTTP/1.1 404 File not found`
: If the requested file is not available

`HTTP/1.1 200 OK`
: If the requested file is available

In all cases no response headers should be sent. In the first two cases, the message body should be empty. In the last case, the message body should contain the requested file.

Once the response message is sent, our simple web server closes the connection.

1.1 The Network Module

A network module (`network.h, network.c`) are provided to handle the networking parts of the simple web server. The network module provides three functions:

`void network_init( int port )`
: This function is called once, when the web server starts up. It takes the port number to which the web server is to bind, and performs all network initialization.

`void network_wait()`
: This function waits until a client is available for the server. If a client is waiting to connect, this function returns immediately. Otherwise, it blocks (puts the web server to sleep until a client is available.

`int network_open()`
: This function opens a connection to the next waiting client and returns a file descriptor. If no client is waiting, this function returns -1.

A full description of these functions is found in `network.h`.

2 Task 1: The Simple Web Server (`sws`)

Your task is to implement the simple web server (sws). The source file `sws.c` should contain the `main()` function. The simple web server should do the following:

1. Take one (1) argument on the command line: The argument a nonnegative integer denoting the port number to which the web server should bind. For example, the invocation

   ```
   ./sws 12345
   ```

   instructs the web server to bind to port 12345 on start up.
2. Initialize the network module.
3. Enter an infinite main loop that
   (a) Waits for one ore more clients to connect.
   (b) Connects to each client and processes their request.
4. Each client's request is processed by following steps:
   (a) Receive the client's request message.
   (b) Parse the client's request message and extracting the requested file path.
   (c) Send back an appropriate response depending on the well formedness of the request and the availability of the file.
   (d) Close the connection.

You can use the provided `makefile` to build your web server simply by running `make`. You will need to supply a `sws.c` file. Please see Section 3 on how you can test your implementation. Note: the sample solution is approximately 100 lines of code.

3 Task 2: Test Your Code

You will need to test your simple web server implementations. You can view the output of your simple web server by using `nc` to send requests to the web server and look at the responses or redirect them to a file.

To test your implementation, you first create an input file containing a request message to your server. Second, you run your web server either in the background or in another terminal window. Suppose your example was contained in the file `test1.in` and your server was bound to port 8080. To view the response of your server you would use the command line

```
cat test1.in | nc localhost 8080
```

and to save the response of your web server in a file called `test1.out` you, you would use the command:

```
cat test1.in | nc localhost 8080 > test1.out
```

You can then examine the output to see if it is as it should be.

Develop 5 test cases to test your simulator. Create three (3) files for each test:

`test.#.in` is the request file containing a request that tests a function of the web server.
`test.#.txt` is a brief description of the test and the expected result.
`test.#.out` is the expected output of the test.

where ## represents a number 0 . . . 4. The `.out` file can be created from the `.in` file. E.g.,

```
cat test_0.in | nc localhost 8080 > test_0.out
```

The tests should test all the functionality of the simulator. The test description files **must** contain: a title, author (your name) purpose of the test, how the test works, and a description of the expected result.

4 Task 3: Written Work

Answer the following questions. Your answers should provide sufficient detail and justification but need not be more than a paragraph long for each part.

1. How could you implement your web server without using the `network_wait()` function?
2. Why is the `network_wait()` function needed?
3. Is there a limit to the size of the file that can be sent? Why?
4. What happens to all the waiting clients if the current client requests a large file (say a few gigabytes in size)?
5. Why is it important to close the file after it is sent?
6. Why is it important to close the connection after the response is sent? (You may need to do some research to answer this question.) Hint: There are at least two reasons.

5 Hints and Suggestions

1. Read up on how to perform file operations in C, you will need them.
2. Be sure to create useful tests. They are worth significant marks and will catch many errors.
3. Start early, the solution totals less than 100 lines, but if you are not comfortable with C or unix, it may take you a bit longer to implement and test.
4. Even if you do not finish the code, submit the test files, they are worth many marks.

6 What to Hand In

You must submit an electronic copy of your assignment. Submissions must be done by 12 noon of the due date. The submission should be done using SVN. See the course web-page (Assignments or Resources) on how to submit using SVN.

6.1 Programming Guidelines

1. Use the file and function names specified in the assignment.
2. Your programs will be compiled and tested on the `bluenose.cs.dal.ca` Unix server. In order to receive credit, your programs must compile and run on this server.
3. All submitted programs must compile. **Programs that do not compile will receive a 0.**
4. All submitted programs must be built with the provided makefile. **Programs that do not build with the makefile will receive a 0.**
5. The code must be well commented, properly indented, and clearly written. Please see the style guidelines under Assignments or Resources on the course website.

6.2 Required Files

Use SVN to submit the following files.

1. The source code files that implement the simple web server.
2. The makefile used to build your web server
3. The test files specified in Section 3
4. The file `questions.pdf` or `questions.txt` containing the answers to the questions in Section 4

7 Grading

	Mark
Simple Web Server	/25
Structure and Design of Web Server	/15
Functionality (automated tests)	/10
Test Suite	/10
Quality (effectiveness of the tests)	/5
Number of tests	/5
Written Work	/10
Question 1	/2
Question 2	/1
Question 3	/2
Question 4	/1
Question 5	/2
Question 6	/2
Code Clarity	/5
Total	**/50**