University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	**NS3**			
Coursework Number	**Summative Exercise 1**			
Deadline	**Time:**	**4:30pm**	**Date:**	**7 March 2014**
% Contribution to final course mark	**16%**			
Solo or Group ✓	**Solo**	✓	**Group**	
Anticipated Hours	**12**			
Submission Instructions	**Submit via Moodle, in a .tar.gz archive formatted as instructed in the NS3 Lab 2 hand-out.**			
Please Note: This Coursework cannot be Re-Done				

Code of Assessment Rules for Coursework Submission

Deadlines for the submission of coursework which is to be formally assessed will be published in course documentation, and work which is submitted later than the deadline will be subject to penalty as set out below.

The primary grade and secondary band awarded for coursework which is submitted after the published deadline will be calculated as follows:

(i) in respect of work submitted not more than five working days after the deadline
   a. the work will be assessed in the usual way;
   b. the primary grade and secondary band so determined will then be reduced by two secondary bands for each working day (or part of a working day) the work was submitted late.

(ii) work submitted more than five working days after the deadline will be awarded Grade H.

Penalties for late submission of coursework will not be imposed if good cause is established for the late submission. You should submit documents supporting good cause via MyCampus.

Penalty for non-adherence to Submission Instructions is 2 bands

You must complete an "Own Work" form via
https://webapps.dcs.gla.ac.uk/ETHICS for all coursework
UNLESS submitted via Moodle

NS3 Lab 2 – Web Server

Dr Colin Perkins
School of Computing Science
University of Glasgow
http://csperkins.org/teaching/ns3/

4/5 February 2014

Introduction

The laboratory exercises for Networked Systems 3 (NS3) will introduce you to network programming in C on Unix/Linux systems, and help you understand the operation of the network. There are weekly laboratory sessions for this course, during which you will complete several exercises. These exercises will build on your knowledge of C programming from the Advanced Programming 3 (AP3) course last semester, and on the material in the NS3 lectures. There are a mixture of formative and summative exercises. The formative exercises are intended to give you practice in programming networked systems in C; they are not assessed. The two summative exercises test your ability to program networked systems.

This is NS3 lab 2, an exercise to build a web server in C. It comprises one summative exercise that should be completed during the timetabled laboratory sessions in weeks 4–8 of the semester, and during other hours as necessary. This work is assessed, and is worth 16% of the marks for this course.

The first version of your server will support only the minimum parts of HTTP required to send a single response to the client; as you proceed further through the exercise you will add more HTTP features, and add support for multiple requests and multiple simultaneous clients. You should attempt all parts of this exercise. It is recommended that you read this entire handout carefully, and think carefully about, and plan, your system design, before you start coding.

The HyperText Transport Protocol

A web browser uses the HyperText Transport Protocol (HTTP) to retrieve pages from a web server. The browser makes a TCP/IP connection to the web server, sends an HTTP request for the requested web page over that connection, reads the response back, and then displays the page. Both HTTP requests and responses are text-based, making the network protocol relatively straight-forward to understand.

An HTTP request comprises a single line command (the "method"), followed by one or more header lines containing additional information. To retrieve a page, a web browser uses the `GET` method, specifying the page to retrieve and the version of the HTTP protocol used (the current version is `HTTP/1.1`). For example, a browser would send the method `GET /index.html HTTP/1.1` to retrieve the page `/index.html` from a server. The `GET` request must be followed by a header to specify the name of the web site, for example `Host: www.gla.ac.uk` (in case there are several sites hosted on the same server). The headers are followed with a blank line, to indicate the end of the request. For example, to fetch the main University web page (`http://www.gla.ac.uk/index.html`), a browser could make a TCP/IP connection to `www.gla.ac.uk` port 80, and send the following request:

```
GET /index.html HTTP/1.1
Host: www.gla.ac.uk
```

Note that each line ends with a carriage return (‘\r’) followed by a new line (‘\n’), and the whole request is terminated by a blank line (i.e., a line containing nothing but the \r\n end of line marker). The example above is a minimal HTTP request. A web browser will usually include many other header lines, in addition to the `Host:` header, to control the connection, indicate support for particular file formats and languages, convey cookies, and so on.

When it receives an HTTP GET request for a web page that exists, a web server will reply with a `HTTP/1.1 200 OK` response, followed by several more header lines providing information about the response, a blank line, and then the body of the page. The headers lines should include a `Content-Length:` header, which specifies the size of the body of the page in bytes. As with the request, each header line ends with a carriage return followed by a new line, and the headers are separated from the body with a blank line. An example of the type of response that is sent follows (“...” indicates that some text has been elided):

```
HTTP/1.1 200 OK
Date: Tue, 12 Jan 2010 11:18:30 GMT
Server: Apache/1.3.34 (Unix) PHP/4.4.2
Last-Modified: Tue, 12 Jan 2010 09:59:31 GMT
ETag: "1a-3d4e-4b4c4803"
Accept-Ranges: bytes
Content-Length: 15694
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
...
</body>
</html>
```

In this example, the “Content-Length:” is 15694 bytes, meaning that there are exactly 15694 bytes in the body of the response (starting with the “<” of the “<!DOCTYPE” line, and finishing with the “>” of the “</html>” line.

If a request is made for a non-existing file, the server will respond with a 404 “file not found” error. This will have a “Content-Type:” header of “text/html”, and the body of the response contains the error page to be displayed to the user.

```
HTTP/1.1 404 Not Found
Date: Tue, 20 Jan 2009 10:31:56 GMT
Server: Apache/2.0.46 (Scientific Linux)
Content-Length: 300
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<head>
<title>404 Not Found</title>
...
</body>
</html>
```

Other types of response are possible, distinguished by the numeric code in the first line of the response.

Summative Exercise 1: A Simple Web Server

The aim this exercise is to write a simple web server. The web server should bind to port 8080 (access to port 80 is restricted to software installed by the systems administrator, so we'll use port 8080 for this exercise), listen for HTTP requests sent by a web browser, and return appropriate responses to the client (either the web page requested, or an error message).

Basic Connection Handling

Your server should create a TCP socket, bind it to port 8080, then listen for and accept connections from browsers. On accepting a connection from the browser, your server should read and parse the request. The first line of the data read from the socket (up to the initial `\r\n`) determines the type of HTTP request being made by the browser. If the request begins `GET` followed by a filename and `HTTP/1.1` then you should parse that request to retrieve the name of the file requested. The filename should be interpreted as being relative to the directory in which your server was started (i.e., if the server was run from directory `/users/staff/csp` and received the request `GET /index.html HTTP/1.1`, it would return the contents of `/users/staff/csp/index.html`). Your server should also check the value of the `Host:` HTTP header sent by the client, to ensure it matches the current hostname (use the `gethostname()` function to find the hostname; be sure to check both *hostname* and *hostname*.dcs.gla.ac.uk – or the equivalent domain name in Singapore). Note that different browsers can send the HTTP header lines in different orders; your code should not assume that the `Host:` header is in a fixed location. After parsing the request to determine the filename, and checking the `Host:` header, your server should respond with the appropriate HTTP headers, followed by the data (the contents of the file). It should then close the connection.

If the hostname matches, and the requested file exists, a success ("200 OK") response should be sent, followed by the contents of the requested file, then the connection should be closed. An example of a minimal successful response, returning an HTML page, is as follows:

```
HTTP/1.1 200 OK
Content-Type: text/html
Connection: close

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
...
```

(the "..." indicates that the output has been truncated in these notes – your server should return a complete web page). The HTTP header lines and the blank file following them *must* be generated by the server for each page: only the HTML page content is read from the file.

If the requested file doesn't exist, a "404 File Not Found" response should be generated. An example "404 File Not Found" response is as follows (the headers indicate that an error has occurred, body of the response is displayed by the browser).

```
HTTP/1.1 404 Not Found
Content-Type: text/html
Connection: close

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
```

```
          "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title> 404 Not Found </title>
</head>
<body>
<p> The requested file cannot be found. </p>
</body>
</html>
```

If the hostname of the server doesn't match the `Host:` header, if the `Host:` header is not present, if the request does not start with `GET`, or if your server doesn't understand the request for some other reason, you should send a "400 Bad Request" response. If your server fails for some other reason, it should send a "500 Internal Server Error" response. Note that the HTTP standard requires the method ("GET") to be in upper case, but the other header lines are case insensitive.

The first line of the response indicates the version of HTTP used (`HTTP/1.1`) and if the request succeeded (`200 OK`) or failed (e.g., `404 Not Found`). This is followed by several header lines giving information about the response, a blank line, and the actual data requested. Two headers are essential: `Content-Type:` tells the browser the format of the data – the first version of your server should use `text/html` for everything – and `Connection: close` tells the client that you will close the connection after sending the data.

Write a simple `index.html` file, and test your server by retrieving this file using a web browser of your choice. The URL you need to give to your browser will depend on the host you're using. For example, if you are using host `bo720-1-01.dcs.gla.ac.uk` to develop and test your server, connect to `http://bo720-1-01.dcs.gla.ac.uk:8080/index.html`. Check that your server correctly responds to requests for both valid and non-existing pages, and that you can see your test page displayed in the browser.

Handling Multiple Sequential Connections

You will recall that the `accept()` function returns a file descriptor for the newly open connection, leaving the file descriptor of the listening socket untouched. A server may therefore accept a new connection, read the request, send its response, and close the connection, all without disturbing the listening socket. Extend your web server to use this feature to accept and serve multiple connections, one after the other, rather than exiting after serving a single connection. Don't forget to set an appropriate backlog in the `listen()` call, so multiple connections can be waiting.

Write a simple web site, comprising multiple HTML pages, for your server to host. Browse this site using your favourite web browser to check that your server correctly responds to multiple requests.

Specifying the Content Type

Extend your website to include some images, in both JPEG and GIF format, linked from the HTML pages. To make the browser recognise these images, you'll need to include an appropriate `Content-Type:` header in the response sent by your server. The `Content-Type:` should be chosen according to the extension of the filename:

Filename:	Content-Type:
*.html, *.htm	`Content-Type: text/html`
*.txt	`Content-Type: text/plain`
*.jpg, *.jpeg	`Content-Type: image/jpeg`
*.gif	`Content-Type: image/gif`
(unknown)	`Content-Type: application/octet-stream`

You'll need to parse the filename in the HTTP GET request to determine the extension, and then fill in the `Content-Type:` appropriately when constructing your response.

Handling Multiple Requests per Connection

Forcing a web browser to open a new TCP connection for each request is inefficient when multiple files are retrieved from a single web server. To avoid this inefficiency, HTTP allows several requests to be sent on a single connection. If the server *does not* include a `Connection: close` header in its response, the client can keep the connection open, and may send additional HTTP requests to the server. To allow the client to distinguish data from multiple requests, the server must include a `Content-Length:` header specifying the size of each response's data in bytes. You can retrieve the size of a file using the `fstat()` function:

```
#include <sys/stat.h>
...
struct stat    fs;
int            fd = open(filename, O_RDONLY);
...
if (fstat(fd, &fs) == -1) {
    // Error...
}
printf("file size = %d\n", fs.st_size);
```

Update your web server to support multiple requests per connection, only closing the connection when the client does so (the `read()` function will return zero when the connection is closed). Demonstrate that this works using a standard web browser by printing details of each request handled by the server.

Handling Multiple Connections in Parallel

Extend your web server to accept multiple connections in parallel. This should improve its scalability and response time on multicore systems, and on systems with relatively slow disks compared to the speed of their network connection. There are two parts to this task: introducing concurrency to your web server by starting a new thread for each connection, and then extending the server with a thread pool, to hide thread creation overheads. Both parts should be implemented using the pthreads API functions.

One thread per connection

Introduce concurrency by extending your server so that it starts a new thread to process each network connection accepted on a socket. That is, when the `accept()` function completes, start a new thread to process the newly accepted connection. The new thread should accept the file descriptor for the new connection as a parameter in the `pthread_create()` call. Once created, the thread will process HTTP requests until the connection is closed by the client, then it will close the connection socket, and exit. The main thread should remain open and accepting new connections.

Test your system to demonstrate that it can successfully handle multiple connections in parallel. A web browser should open multiple connections to a site if you have enough content for it to fetch. Create a web page containing several (at least 10) images, and test concurrent browsing using this page (print the thread identifier, returned by `pthread_self()` and filename when returning a response, to shows that it's working in parallel).

Thread pool

Starting a new thread for each connection can be inefficient, since threads take some time to start. A more scalable approach creates a pool of worker threads before accepting any connections, and passes each new connection to the next free thread in the pool. Such a system comprises a single controller thread, with a pool of worker threads. The controller maintains a list of workers, along with a state variable for each indicating if it is busy of idle. Newly accepted connections are passed to the first idle worker in the list, with new connections not being accepted until there is an idle worker thread.

Implement such a system using pthreads to create the worker threads. Take care to provide appropriate locking when manipulating condition variables shared between controller and workers. Ensure your workers block on the condition variable while waiting for new work, rather than continually polling. Test your system, to demonstrate that it works correctly.

Submission

Your server must be written in C, and must run on the Linux machines in the level 3 laboratory. You are *required* to write a simple Makefile to compile your code, rather than running the compiler by hand. You are also *strongly advised* to enable all compiler warnings (at minimum, use `gcc -W -Wall`), and to fix your code so it compiles without warnings. Compiler warnings highlight code which is legal, but almost certainly doesn't do what you think it does. Use them to help you find problems.

You should prepare an electronic copy of your source code and Makefile (do not submit compiled binaries) archived as a `.tar.gz` file that expands into a directory named after your 7-digit matriculation number followed by "-submission1". For example, if your matriculation number is 0301234, your archive should expand to create a directory "0301234-submission1" with your files inside. You can create the archive using a command such as:

```
tar cvzf 0301234-submission1.tar.gz 0301234-submission1/
```

Ask one of the lab demonstrators if you are unsure how to create the archive. If your archive is formatted correctly, you should see something like the following when running the `tar ztf` command:

```
$ tar ztf 0301234-submission1.tar.gz
0301234-submission1/Makefile
0301234-submission1/web-server.c
$
```

(the `0301234-submission1/` prefix shows that the archive expands into a subdirectory with the appropriate name for this matriculation number).

This work is assessed, and is worth 16% of the marks for this course. Submissions should be made via Moodle. The deadline for submissions is 4:30pm on Friday 7 March 2014 As per the Code of Assessment policy regarding late submissions, submissions will be accepted for up to 5 working days beyond this due date. Any late submissions will be marked as if submitted on time, yielding a band value between 0 and 22; for each working day the submission is late, the band value will be reduced by 2. Submissions received more than 5 working days after the due date will receive an H (band value of 0). Submissions that are not made via Moodle, or that are in archives which do not meet the above guidelines will be penalised two bands. This penalty will be applied in addition to any late submission penalty.