COSC1176 / 1179
Network Programming

Assignment 2
Web Server

GENERAL INFORMATION

The assignment must be performed by each student individually.

This assignment is marked out of 100. It is worth 30% of your overall overall course score.

Submission is due Friday 7th October at 10:00 PM.

You may use C or C++. Demos and marking will be performed on *yallara*.

Some sections of this document are marked "*Consider: ...*" These sections are not specifically marked but you might find it helpful to think about the issues mentioned.

PLAGIARISM NOTICE

You must write all code yourself. You may not use any external source code, e.g. code from the internet. You can, of course, use and modify the basic examples given as part of the course.

Plagiarism (submitting any code from the external sources or sharing any work amongst students) will not be tolerated. Any student suspected of plagiarism will be referred to the School's higher authorities. More details can be found here.

OUTLINE

This assignment is in 5 parts. It involves writing 4 versions of a web server, and modifying your gethhtp so you can test and demonstrate your servers. The 4 servers are: single process, multi-processing, multi-threaded and multiplexed I/O.

The assignment requires you to research and understand HTTP well enough that your server can reliably serve current web browsers (e.g. Internet Explorer, Google Chrome, Firefox) and serve your gethhtp client, using HTTP 1.0 and HTTP 1.1. The official description of the protocols is the RFCs (including RFC 1945 and RFC 2616), you may use other appropriate sources to help understand HTTP (e.g. textbooks, Internet sites, etc) but all code must be your own.

We recommend that you implement only that part of the HTTP protocol than is required by the assignment. There are no additional marks for additional functionality, and additional functions will require a lot of additional effort – so don't implement transfer coding, content coding (chunked data), multi-part (MIME) content, content negotiation, authorisation, or persistent connections.

Appendix 1 gives details of the required functions of your web server. Your web server only needs to serve GET requests – it does not need to fully implement other methods, like POST or HEAD.

INSTRUCTIONS

Your work on assignment 1 has provided code and understanding for most of this assignment. Reuse your config-file and command-line code in this assignment – if necessary, extend it and tidy it up. Your tutes/labs have shown you the basics of multi-processing/threading, you can reuse that code – but be sure you handle zombies sensibly. Assignment 1 has given you practice in HTTP and using the looping structures needed to read/write sockets/files – use similar structures for your servers.

We recommend very strongly that you first get all of the necessary server functions operating (reliably and fully debugged) on the single-process server. Then extend this server into multi-processing, multi-threaded, and multiplexed I/O. If you are part-way through one of the concurrent servers, and find a bug in basic functionality, then you will need to go back and fix the bug in each server. It is much easier to find the bugs once, before you have several servers to fix.

SUBMISSION AND DEMONSTRATION

Submission details will be advised closer to the due date. You may organise your code in any sensible manner, so you might submit a single set of source files, or might submit five separate source trees under one home directory. Your submission must use 'make' to build programs called: `server-single`, `server-forked`, `server-threaded`, `server-select`, and `slow-gettp`.

You will be asked to:

- Compile parts 1-5 using the make command "`make all`".
- Demonstrate that each version of your server can serve multiple concurrent copies of your slow-gethttp program. You must use the supplied script `get5now` to concurrently execute 5 instances of `./slow-gehttp` to request the files `test1.dat` – `test5.dat` from your server.
- Demonstrate that each version of your server handles web requests from several concurrent sessions of a standard web browser.

ASSIGNMENT DETAILS

Part 1 – Initial implementation – server-single (20 marks)

Implement and debug "server-single", a web server that performs all required web server functions without concurrency (single process, single thread). This server should accept a single request and process it to completion, before accepting the next request. More detail about the functions that the server should perform are given in appendix 1.

Save a copy before moving onto parts 2-4, as 20 marks are available for demonstrating basic functionality.

Part 2 – Write a slow-gethhtp client – (not marked)

Write "slow-gethhtp" by modifying your gethhtp client from assignment 1, so that it receives the HTTP content with a delay of 1 second after reading each buffer, and has a buffer size of 1 kB.

This part of the assignment is not marked, but is required for you to test and demonstrate the other parts. Your slow-gethhtp must be part of your submission.

Your slow-gethhtp client should have a conventional read/write loop. You need only insert a delay into the loop. The basic loop will function something like this pseudocode:

```
...
while (readSz ← read(socket, buffer, bufSize)) > 0
      print srcFileName + ": got " + readSz + " bytes from server"
      if write(file, buffer, readSz) ≠ readSz
            // handle the write error
      sleep(1)
...
```

Part 3 – Implement concurrency using multiple processes – server-forked (20 marks)

Write "server-forked" by modifying your web server from part 1 to handle at least 10 concurrent requests using multiple processes. Your server should fork( ) a new process for each connection request.

Consider: How will this server perform with a realistic clients?

You must be able to demonstrate that your server is handling requests concurrently. Your server should log the request and completion time for each connection (see the appendix) – this will help you to see that multiple connections were served concurrently.

Use your slow-gethttp client to test and execute several concurrent http connections. You might also generate concurrent requests using multiple browser sessions, if you are quick and lucky. You might find that your standard browser performs multiple concurrent requests.

Your server must handle processes correctly (e.g. avoid zombies). After a shutdown request, your server must stop accepting new requests and wait until all existing requests have completed, before the main process terminates.

Consider:

- *How is a child processes going to tell the server (main) process that a shutdown has been requested?*
- *How are you going to keep track of the number of child processes, so the main process does not exit before all current requests are finished?*

Part 4 – Implement concurrency using multiple threads – server-threaded (20 marks)

Write "server-threaded" by modifying your web server-single from part 1 to handle at least 10 concurrent requests using multiple threads. Your server should spawn a new thread for each connection request.

Your server must be able to demonstrate requests being handled concurrently. Your server log should include, for each connection, the accept and close/shutdown times (see the appendix).

Your server must handle threads correctly (e.g. avoid zombies). When shutting-down, your server must stop accepting new requests and wait until all existing requests have completed before terminating.

Consider:

- *How are you going to keep track of the number of child threads?*
- *How are you going to handle thread completion (prevent zombies)?*

Part 5 – Implement concurrency using multiplexed I/O and select( ) – server-select (30 marks)

Write "server-select" by modifying your the web server from part 1 to handle at least 10 concurrent requests using multiplexed I/O with select( ).

Your server must execute as a single process with a single thread (a normal process). It should not block if any connection can proceed. It should never "spin waiting". Use select( ) to determine which sockets/files are ready, and to wait efficiently.

Your server must be able to demonstrate several requests being handled concurrently. Use your slow-gethttp client to test and demonstrate several concurrent get-hhtp sessions. You will also need to demonstrate multiple concurrent sessions using a standard browser.

When shutting-down, your server must stop accepting new requests and wait until all existing connections have completed before exiting.

Consider:

- *How are you going to keep track of the state of each connection?*
- *How will you know what actions to perform next, for each connection?*
- *Will each connection have its own buffer, or will connections share buffers?*
- *How are you going to build the FD_SETs for select( )? How do you know if a connection is waiting to read the socket, write to the socket, or read from the file?*

Additional marks for parts 1-5 – Code quality (10 marks)

Your code should be robust, safe, readable and efficient.

Remember to check the return value of system calls, use 'errno'.

Have meaningful function and variable names, readable code and clear, useful comments.

Your program should never crash, or have "memory leaks".

Your program should compile and pass through 'lint' without errors or warnings.

APPENDIX 1 – REQUIRED FUNCTIONS FOR WEB SERVER

The following functions are required for your servers – that is for parts 1, 3, 4 and 5.

For testing, we will provide a sample config file, sample "root" directory, and a get5now script. For the demo, your server is expected to operate with a similar (but different) config file and directory.

Your server should be able to serve all the pages/files under the root directory, irrespective of file extension, content type, or file size. Implementing only the HTTP GET method should be sufficient for this purpose.

You should be able to browse the test pages and access links normally, using a normal browser, by giving a URL like `goanna.cs.rmit.edu.au:54321/index.htm` (use your own port number).

Your server should behave sensibly when given invalid or incorrect results. For example, if given a request you have not implemented (e.g. a POST or HEAD request), your server should return a HTTP reply with an appropriate status code.

Your server should function sensibly when connections terminate unexpectedly. It should clean up that connection's process/thread/state and continue all other processing normally. The log file should include one line for each connection – after a connection error, the log should give the errno code or a description of the error.

At all times, your server should behave like a professional server: it should not crash, leak memory, send badly formed or inappropriate HTTP replies or status codes, or reject reasonable requests..

Your server should use a config file for all required control

Extend your config file handling from assignment 1, as required. The server must require and handle a command line like:

```
server-single test.config
```

Your server should handle GET requests from both HTTP 1.0 and HTTP 1.1 clients

Note that HTTP 1.0 has a slightly different request format to HTTP 1.1.

The config file includes one line, specifying the serving (listening) port, like:

```
port 56789
```

Your server should serve files from a local root directory, the root is specified in the config file

The config file includes one line like:

```
root /dirpath/test-dir              # or perhaps   localdir/mytest-dir
```

Your server should return sensible HTTP status codes for all requests

For example, you should return codes 200 (success), 204 (success but no data), 404 (not found), and your server should also be capable of returning 400 (bad request), 405 (method not allowed), 408 (request timeout) and 501 (not implemented) when these or similar codes are appropriate.

Your server should operate with current versions of the following clients:

You should be able to download any file using your gethttp or slow-gethttp client. You should be able to browse the test pages and links using a normal browser: e.g. Internet Explorer, Firefox, Google Chrome.

Your server's HTTP replies should include Content-length information

Your server should correctly report the exact size of the data to be returned, using the Content-length header field. *(What Unix system call will tell you the size of the file?)*

Your server's HTTP replies should indicate that the server is not providing persistent connections.

Your server should correctly include a "`Connection: close`" header field.

Your server's HTTP replies should include a date header.

Your server should correctly include a "`Date:`" header field.

Your server's HTTP replies should include a server header.

Your server should correctly include a "`Server:`" header field that describes your server.

Your server's HTTP replies should include Content-type information using the filename extension to decide the content type. The mapping from file extension to content type is given in the config file.

The config file includes lines, for most of the files that might be found, like these. For the first line, where the file extension is "`.txt`", the corresponding HTTP content type is "`text/plain`". The last setting ("`type`") is the default content-type that you should use when you are serving a file where you don't recognise the file extension.

```
type-txt text/plain
type-htm text/html
type-html text/html
type-jpg image/jpeg
type-mp3 audio/mpeg
type-wav audio/vnd.wave
type text/plain
```

Your server does not have to perform content negotiation, but should correctly report the content type of the file requested.

Your server may ignore or refuse options that are not required to serve the requests, e.g. content type negotiation, connection reuse, ...

Most request options can safely be ignored. To refuse a request, return a 501 status in your reply.

Your server does not have to forward "proxy" requests

The config file includes one line, specifying the local server's name, like:

```
host goanna.cs.rmit.edu.au
```

If receiving a request for a different host machine, your server should return a suitable status code indicating that it cannot fulfil the request (e.g. status 305).

Your server should implement full logging (controlled by the config file)

The config file includes lines like:

```
logfile mydir/webserver.log
logging yes                          # or perhaps no
```

The log file should include at least the following information, where applicable. For every request, including the shutdown requests, emit the following:

- connection accepted date
- connection accepted time
- connection closed time
- client's IP address
- client's TCP port
- pathname of requested file
- status code returned to client
- length of content (in bytes)
- errno code or description for connection error (use 0 for "no error")

For example:

```
30/9/2011 22:30:20 server-forked 1.2.2
30/9/2011 22:30:21 initialization complete
30/9/2011 22:30:21 listening on port 56789
30/9/2011 22:33:44 22:33:44 123.234.210.101 4321 /control/status/icon.gif 404 0 0
30/9/2011 22:35:05 22:35:06 123.234.210.101 4322 /test/icon.gif 200 260 0
30/9/2011 22:35:02 22:35:23 123.234.210.101 4320 /test/2big.pdf 200 123456789 104
30/9/2011 22:35:22 22:35:23 123.234.210.101 4323 /test/2big.pdf 200 123456789 0
30/9/2011 22:35:12 22:35:12 123.234.210.101 4328 /control/shutdown.htm 200 0 0
30/9/2011 22:35:13 shutdown request
30/9/2011 22:35:14 22:35:14 123.234.210.101 4325 /test/small.gif 503 0 0
30/9/2011 22:35:10 22:35:18 123.234.210.101 4344 /test/medium.pdf 200 456789 0
30/9/2011 22:35:20 all connections closed
30/9/2011 22:35:20 terminating server
```

Your server should implement full request recording (controlled by the config file)

The config file includes lines like:

```
recordfile mydir/request.txt
recording yes                              # or perhaps no
```

If enabled, the record file should contain an exact copy of the last request received. You will find this very useful for debugging.

Your server must perform controlled shutdown triggered by any of: a specific interrupt given in the config file, or a request for a specific file (which might not exist) named in the config file.

The config file includes lines like:

```
shutdown-signal 15
shutdown-request config/shutdown.htm
```

Controlled shutdown requires the server detect the shutdown request, log the request, refuse new connections, continue processing any existing requests, wait until all existing requests have been completed and those processes/threads/state have been tidied-up, log the shutdown, and finally exit.

Consider:

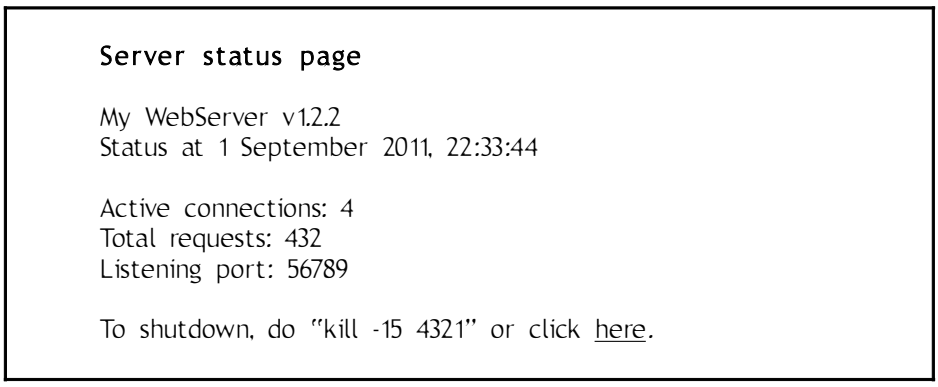
- *How are you going to refuse connection requests?*

Generate a basic status page at the address named in the config file.

The config file includes a line like:

```
status-file config/status.htm
```

All requests for this file should return some simple HTML text (as if a file exists with the correct contents). The HTML text should include the following information: current date, current time, number of active connections, the total number of requests handled, the server port, and a link to the shutdown URL. (For the single process server, the number of active processes will only ever be 1.) For example:



Sample config file.

The config file includes lines like:

```
# web server config

# use one of your allocated ports
# change 'host' to actual machine name
port 56789
root ass2/test
host goanna.cs.rmit.edu.au

# how to shutdown the server
shutdown-signal 15
shutdown-file config/shutdown.htm
status-request config/status.htm

# logging and debugging controls
logfile webserver.log
logging yes
recordfile lastrequest.txt
recording yes

# map file extensions to content type
type-txt text/plain
type-htm text/html
type-html text/html
type-jpg image/jpeg
type-mp3 audio/mpeg
type-wav audio/vnd.wave
# default content for null or unrecognised extensions
type text/plain

# end
```

- End of assignment -