COMS W3157 Advanced Programming, Lab #7

Please refer to our instructions for lab retrieval and submission. You must include the required information in your README.txt. This lab consists of multiple parts, so your code for each part must be in a subdirectory (named "part1", "part2", etc.). Like before, you are required to have 5 Git commits with meaningful commit messages and a Makefile for each part where you write code.

For this lab ONLY, you are not required to include Valgrind output in your README.txt. You should keep using Valgrind to check for memory errors (and TAs will do the same when grading), but you do not have to submit the output.

Part 1: mdb-lookup-server

Your job is to write mdb-lookup-server, a program that does the same thing as mdb-lookup from Lab 4, except that it communicates with clients over a TCP/IP connection rather than stdin/stdout.

mdb-lookup-server takes the following arguments:

```
    mdb-lookup-server <server-port> <database>
```

mdb-lookup-server listens on <server-port> for TCP connections, and searches through the <database> file according to each client's search queries. mdb-lookup-server should support multiple simultaneous client connections by handling each client in a separate process.

This part is an exercise in adapting and combining existing code; there is actually very little new code you need to write, as long as you can read and understand code from prior labs. You should combine the mdb-lookup source code from Lab 4 with the multi-process TCP server architecture from Lab 6 part 2.

As usual, here are some hints and requirements:

- Your server must be implemented in a file named mdb-lookup-server.c and build to an executable named mdb-lookup-server. You should #include the provided mdb.h header file, which contains the struct MdbRec definition.

  Like Lab 4, you must use the mylist library from Lab 3 to store mdb records. You MUST use the header file and library archive we have installed in /home/j-hui/cs3157-pub/include and /home/j-hui/cs3157-pub/lib, respectively. Do NOT use your own Lab 3 solutions.

- Upon accepting a client connection, mdb-lookup-server should fork() a child process to handle the connection, which will then load the <database> file into a linked list before entering the mdb-lookup loop like in Lab 4. Meanwhile, the parent process should continue to accept() other client connections.

  Note that mdb-lookup-server should load the <database> file once per client connection; if the <database> file is updated after mdb-lookup-server is started, subsequent clients should see those changes.

- mdb-lookup-server conveys search results using a simple text-based protocol. First, the client will send the server a line of text containing the search query. mdb-lookup-server will search its database using up to the first

5 characters of the search query (just like in Lab 4), and send the formatted search result lines to the client (in the same format as Lab 4). Finally, mdb-lookup-server will send a blank line to the client to signal that it has sent all the search results for that query.

Note that, unlike mdb-lookup from Lab 4, mdb-lookup-server should NOT send the "lookup:" prompt to the client.

- mdb-lookup-server should be able to handle search query lines of any length, terminated by either "\r\n" or "\n" (and those characters should not be used to search database records). However, mdb-lookup-server should only respond with lines ending with "\n", NOT "\r\n".

- The client may reuse the same TCP connection to send more search queries. So, after sending one round of search results, the server should not close the TCP connection if the client is still connected; instead, it should continue reading search queries from its client.

  Once the client closes the connection, the server process handling that connection should then close its side of the TCP connection, free all its allocated resources, and exit.

- Each handler child process of mdb-lookup-server should log two lines of output to stderr: once when the client connection is first established, and once after the client connection is terminated. They should contain the client's IP address, using the following format:

  ```
  $ ./mdb-lookup-server 8888 my-mdb
  Connection started: 128.59.15.27
  Connection terminated: 128.59.15.27
  ```

  Note that lines logged for different connections may be interleaved, e.g.:

  ```
  $ ./mdb-lookup-server 8888 my-mdb
  Connection started: 128.59.15.27
  Connection started: 127.0.0.1
  Connection terminated: 127.0.0.1
  Connection terminated: 128.59.15.27
  ```

  In this case, the client with IP 127.0.0.1 connected after the client with IP 128.59.15.27, but disconnected before it.

- Like in Lab 6, mdb-lookup-server needs to be resilient to client failure and ignore potential SIGPIPEs. The parent process should also automatically reap terminated child handler processes using a SIGCHLD handler. You should configure these signals' dispositions using sigaction(), prior to listening for any TCP connections.

Part 2: dynamic web server

In this part, you'll extend the (single-process) http-server from Lab 6 part 1 to support performing mdb-lookup from the browser. Your web server will work in conjunction with mdb-lookup-server from part 1 of this lab, using a three-tier architecture: it will defer to mdb-lookup-server to perform each search, and format the search results as an HTML table that it sends to the HTTP client. Since that HTML table is generated on-the-fly, this extension converts Lab 6's static server into a dynamic one.

Your dynamic http-server takes the following arguments:

```
    http-server <http-port> <web-root> <mdb-host> <mdb-port>
```

Your server should listen for connections on <http-port> and serve static files from <web-root> like before, but should also connect to the mdb-lookup-server at <mdb-host> on <mdb-port>.

You can test out the dynamic http-server like this:

(1) Start mdb-lookup-server with the class database file:

```
        $ ./mdb-lookup-server 9999 ~j-hui/cs3157-pub/bin/mdb-cs3157
```

(2) Open up another terminal window to the same machine and run http-server, which will connect to the mdb-lookup-server you started earlier:

```
        $ ./http-server 8888 ~/html localhost 9999
```

(3) Point your browser to the HTTP server:

```
        http://the.host.name:8888/mdb-lookup
```

(4) If you type "hello" into the text box (without quotes) and submit, you will see a few messages containing hello nicely formatted in a HTML table.

    If you look at the browser's location bar, you'll see that your search string, "hello", is now appended to the URL:

```
        http://the.host.name:8888/mdb-lookup?key=hello
```

Here are some hints and requirements for implementing this functionality:

- Your implementation should still build an executable named http-server.

- You must make a TCP connection to the mdb-lookup-server when your http-server starts up, and keep using the SAME socket connection for all search queries. There should only be a single, persistent connection to the mdb-lookup-server throughout the entire execution of your http-server.

  You use that connection to send search strings to the mdb-lookup-server and read back the results. The end of results is demarcated by a blank line.

- http-server should serve static web pages from <web-root>, like before.

  However, when the request URI is just "/mdb-lookup", or begins with "/mdb-lookup?", you should send the client the mdb-lookup form as a webpage.

  When the request URI begins with "/mdb-lookup?key=", you should also extract the search string after "key=" and send it to mdb-lookup-server. Then, construct an HTML table containing the search results, and include that table in the page you send to the client, after the mdb-lookup form.

- When you send the search string to mdb-lookup-server, make sure you append "\n" to it (think about why!).

- Note that for the dynamic mdb-lookup page, if "/mdb-lookup?" is followed by anything other than "key=", you should only serve the mdb-lookup form (without any search results table).

- Here is what the HTML page should look like, without any search results:

```
    <html><body>
    <h1>mdb-lookup</h1>
    <p>
    <form method=GET action=/mdb-lookup>
    lookup: <input type=text name=key>
    <input type=submit>
    </form>
    <p>
    </body></html>
```

  And here is what an HTML page with the search results table should look like, with the search string "ryan":

```
    <html><body>
    <h1>mdb-lookup</h1>
    <p>
    <form method=GET action=/mdb-lookup>
    lookup: <input type=text name=key>
    <input type=submit>
    </form>
    <p>
    <p><table border>
    <tr><td>
       2: {ryan} said {hi mom}
    <tr><td bgcolor=yellow>
     154: {aryana} said {hi}
    <tr><td>
     367: {ryan's dad} said {what about me?}
    <tr><td bgcolor=yellow>
     946: {mom} said {hi ryan}
    </table>
    </body></html>
```

  Note that the table alternates between <tr><td> and <tr><td bgcolor=yellow> to make the rows easier to read.

- Since you don't know how long the lookup result will be, you should build the search result table row by row, using each line from the results returned by mdb-lookup-server.

- You can hard-code fragments of the mdb-lookup form in your C code like this:

```
    const char *form =
        "<h1>mdb-lookup</h1>\n"
        "<p>\n"
        "<form method=GET action=/mdb-lookup>\n"
        "lookup: <input type=text name=key>\n"
        "<input type=submit>\n"
        "</form>\n"
        "<p>\n";
```

  Note that C automatically concatenates adjacent string literals; in the code above, form points to a single, long string literal. You can construct the the mdb-lookup form just by sending hard-coded HTML fragments to the client.

- To make it easier to read lines of text from the mdb-lookup-server, you should wrap your persistent socket connection with mdb-lookup-server in FILE pointers, using fdopen().

Those FILE pointers will remain open for the entire lifetime of your http-server, so when you quit your http-server using Ctrl-C, you'll leak the memory allocated for those FILE pointers! This memory leak is ok, as long as your http-server only leaks a CONSTANT amount of memory: assuming http-server is terminated while it isn't serving any clients, the amount of memory it leaks should be the same every time. In other words, the amount leaked should not grow as it serves more clients.

- Make sure the HTTP request logging from Lab 6 is still working, and that SIGPIPE is still ignored.

Part 3: latency benchmarking (optional)

Recall that mdb-lookup-server only reads in the message database file once per client connection: entries added to the database since a connection is established won't be included in any searches requested by the connected client. If a client wants the most up-to-date search results, it can always reconnect with mdb-lookup-server, causing the new mdb-lookup handler to read in the database file again.

Part 2's http-server does not do anything to avoid outdated search results: it only maintains a single, persistent connection with mdb-lookup-server for the entirety of its execution. While this leads to incomplete search results, it also avoids the overhead of establishing a new mdb-lookup connection for each search query.

In this part, you will quantify that overhead. Inside your part3 directory, create a copy of your http-server implementation, and modify it to establish a new connection to mdb-lookup-server for EACH /mdb-lookup?key= search request.

This modification will introduce additional latency for those search requests. Try performing the lookup from your browser. Do you see any speed difference? Probably not. The magnitude of the mdb-lookup-server connection overhead is dwarfed by other sources of latency and is probably too small for you to notice.

Instead, you will measure the latency using a latency benchmarking tool, http-lat-bench, whose source is included in the bench/ directory of this lab's skeleton files. It takes the following parameters:

```
    http-lat-bench <hostname> <port> <URIs...>
```

For example, you can ask it to connect with localhost on port 8888 and randomly alternate between HTTP requests for /mdb-lookup?key=yo and /mdb-lookup by running it like this:

```
    $ ./http-lat-bench localhost 8888 /mdb-lookup?key=yo /mdb-lookup
```

Try reading the source code of http-lat-bench to understand how it works; you can even modify some of the #define directives to customize its behavior.

Use this tool to benchmark your part 2 http-server, and compare it against the http-server from part 3. Make sure you choose a workload that exercises the server's mdb-lookup functionality. What is the overhead from part 3's modifications, if any? Document your experiments in your README.txt, and discuss your results.

Part 4: optimizing your server (optional)

Can you make your server deliver up-to-date mdb-lookup results like in part 3, while retaining the performance from part 2?

This part is totally open-ended, with the only rule being that your http-server is not allowed to access the database file (you must maintain the three-tier architecture). Document everything your README.txt.

For this part only, you don’t need to stick to previous parts’ requirements, but make sure any optimizations to http-server or mdb-lookup-server are only made in the part4/ directory; whatever you submit in previous parts must still conform to this lab’s requirements.

--

Good luck!