COMS W3157 Advanced Programming, Lab #6 (120 points total)
---

Please read this assignment carefully and follow the instructions EXACTLY.

Submission:

Please refer to the lab retrieval and submission instruction. The requirements regarding subdirectory for each part, README.txt and Makefiles remain the same as the previous labs.

Checking memory errors with valgrind

Do not include valgrind output in README.txt. You should keep using valgrind to check your program for memory error (and the TAs will do the same when grading), but you don't have to include the output in README.txt anymore.

## Part 1: mdb-lookup-server using sockets API

Now that you learned how to use sockets API, it's time to implement a real mdb-lookup-server that does not rely on netcat tool.

Your job is to write mdb-lookup-server.c, a program that does the same thing as mdb-lookup.c from lab 4, except that it communicates with the client via a socket rather than standard I/O streams.

You start the mdb-lookup-server as follows:

```
$ ./mdb-lookup-server my-mdb 8888
```

where my-mdb is the database file name and 8888 is the port number that server should listen on. You can use netcat to connect to the server and use the lookup service.

For this part, it's okay for the server to handle only a single client at a time, so you don't need to fork().

This part is an exercise in combining existing code to make something new. There is actually very little code you need to write. You can basically combine tcp-recver.c that we studied in class and mdb-lookup.c from lab 4 solution. Here is how:

- Every time you get a socket from accept(), you basically run the code for mdb-lookup. That is, you open the database file, read all records into memory, run the lookup loop, and finally clean up resources when the client is done using the lookup service. When you're done with mdb-lookup code, you will go back to accept() in order to service the next client.

The main thing you need to do is to convert the mdb-lookup code to read and write using the socket rather than stdin and stdout.

- For reading, the easiest thing to do is to wrap the socket file descriptor with a FILE*, so that we can leave the fgets() code alone. fdopen() does exactly that:

```
    FILE *input = fdopen(socket_descriptor, "r");
```

will give you a FILE* that you can use in place of stdin.

When you're done, you should fclose(input), which will close the underlying socket for you.

- For output, you need to use send() instead of printf(). Here, the easiest thing to do is to use snprintf() to first print to a char array and then send() what you just printed. Note that, on success, snprintf() returns the number of characters printed.

We will use this mdb-lookup-server in lab 7, where we write a web server which can not only serve static HTML pages, but also serve mdb-lookup results to web browsers. In order to use mdb-lookup-server in conjunction with a web server in lab 7, we need to do two things when we write mdb-lookup-server now:

- Do NOT send the "lookup:" prompt to the client. Simply read the input line (which contains the search word) from the socket, send all the search result lines to the socket, and finally send a blank line to the socket. The blank line at the end is important: the web server will use it to determine the end of a search result.

- Add the following lines in the beginning of your main():

```
    // ignore SIGPIPE so that we don't terminate when we call
    // send() on a disconnected socket.
    if (signal(SIGPIPE, SIG_IGN) == SIG_ERR)
        die("signal() failed");
```

## Part 2: Web page downloader

There is a very useful program called "wget". It's a command line tool that you can use to download a web page like this:

```
wget http://www.gnu.org/software/make/manual/make.html
```

which will download the make manual page, make.html, and save it in the current directory. wget can do much more (downloading a whole web site, for example); see man wget for more info.

Your job is to write a limited version of wget, which we will call http-client.c, that can download a single file. You use it like this:

```
./http-client www.gnu.org 80 /software/make/manual/make.html
```

So you give the components of the URL separately in the command line: the host, the port number, and the file path. The program will download the given file and save it in the current directory. So in the case above, it should produce make.html in the current directory. It should overwrite an existing file.

Hints:

- The program should open a socket connection to the host and port number specified in the command line, and then request the given file using HTTP 1.0 protocol. (See http://www.jmarshall.com/easy/http/ for HTTP 1.0 protocol.) An HTTP GET request looks like this:

```
GET /path/file.html HTTP/1.0
[zero or more headers ...]
[a blank line]
```

- Include the following header in your request:

```
Host: the.host.name.you.are.connecting.to:<port_number>
```

  Some web sites require it.

- Use "\r\n" rather than "\n" as newline when you send your request. It's required by the HTTP protocol.

- Then the program reads the response from the web server which looks like this:

```
HTTP/1.0 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/html
Content-Length: 1354

<html>
<body>
<h1>Happy New Millennium!</h1>
(more file contents)
  .
  .
  .
</body>
</html>
```

  Just like in part 1, you can use fdopen() to wrap the socket with a FILE*, which will make reading the lines much easier.

- The "200" in the 1st line indicates that the request was successful. If it's not 200, the program should print the 1st line and exit.

- After the 1st line, a bunch of headers will come, then comes a blank line, and then the actual file content starts. Your program should skip over all headers and just receive the file content.

- Note that the program should be able to download any type of file, not just HTML files.

- The server will terminate the socket connection when it's done sending the file.

- You will need to pick out the file name part of a file path (make.html from /software/make/manual/make.html for example). Check out strrchr().

- You will need to convert a host name into an IP address. Here is one way
  to convert a host name into an IP address in dotted-quad notation:

```
struct hostent *he;
char *serverName = argv[1];

// get server ip from server name
if ((he = gethostbyname(serverName)) == NULL) {
    die("gethostbyname failed");
}
char *serverIP = inet_ntoa(*(struct in_addr *)he->h_addr);
```

  The man pages of the functions will tell you which header files need to be
  included.

--

Good luck!