CSCI 4210 — Operating Systems
CSCI 6140 — Computer Operating Systems
Homework 4 (document version 1.1)
Sockets and Server Programming using C

Overview

- This homework is due by 11:59:59 PM on Monday, December 4, 2017.
- This homework will count as 8% of your final course grade.
- This homework is to be completed **individually**. Do not share your code with anyone else.
- You **must** use C for this homework assignment, and your code **must** successfully compile via `gcc` with absolutely no warning messages when the `-Wall` (i.e., warn all) compiler option is used. We will also use `-Werror`, which will treat all warnings as critical errors.
- Your code **must** successfully compile and run on Submitty, which uses Ubuntu v16.04.3 LTS. Note that the `gcc` compiler is version 5.4.0 (`Ubuntu 5.4.0-6ubuntu1~16.04.4`).

Homework Specifications

In this final homework assignment, you will use C to write server code to implement a data storage server using server sockets.

For your server, clients connect to your server via TCP. More specifically, clients connect via a specific TCP port number (i.e., the listener port); this TCP port number is the first command-line argument to your server.

Your server must **not** be a single-threaded iterative server. Instead, your server must use either multiple threads or multiple child processes. Your choice!

As with previous assignments, your server must also be parallelized to the extent possible. As such, be sure you handle all potential synchronization issues.

Note that your server must support clients implemented in any language (e.g., Java, C, Python, etc.); therefore, be sure to handle only streams of bytes as opposed to specific language-specific structures.

And though you will only submit your server code for this assignment, feel free to create one or more test clients. Test clients will **not** be provided, but feel free to share test clients with others via Piazza. Also note that you can use `netcat` to test your server.

Application-Layer Protocol

The application-layer protocol between client and server is a line-based protocol (see the next page). Streams of bytes (i.e., characters) are transmitted between clients and your server.

More specifically, clients connect to the server and can add and read files; clients can also request a list of files available on the server.

Once a client is connected, your server must handle as many client commands as necessary, closing the socket connection only when it detects that the remote client has closed its socket.

Use a dedicated child process or child thread to handle each TCP connection.

All regular files must be supported, meaning that both text and binary (e.g., image) files must be supported. To achieve this, be sure you do **not** assume that files contain strings; in other words, do **not** use string functions that rely on the `'\0'` character. Instead, rely on byte counts.

You can assume that the correct number of bytes will be sent and received by client and server. In practice, this is not a safe assumption, but it should greatly simplify your implementation.

Files must be stored in a directory called `storage`. As your server starts up, check to be sure this directory exists; if it does not, report an error to `stderr` and exit. In general, your server can either keep track of stored files via a (shared) structure or use system calls to check for the existence of stored files. Note that you will have synchronization issues to resolve here (e.g., what happens when a client attempts to read a file as it is being written by another client?).

On Submitty, you can assume that the `storage` directory already exists and is initially empty.

Finally, subdirectories are not to be supported. A filename is simply a valid filename without any relative or absolute path specified. More specifically, we will test using filenames containing only alphanumeric and `'.'` characters; you may assume that a filename starts with an alpha character. You may also assume that each filename is no more than 32 characters.

The application-layer protocol must be implemented exactly as shown below:

```
PUT <filename> <bytes>\n<file-contents>
-- store file <filename> on the storage server
-- if <filename> is invalid or <bytes> is zero or invalid,
   return "ERROR INVALID REQUEST\n"
-- if the file already exists, return "ERROR FILE EXISTS\n"
-- return "ACK\n" if successful


GET <filename> <byte-offset> <length>\n
-- server returns <length> bytes of the contents of file <filename>,
   starting at <byte-offset>
-- if <filename> is invalid or <length> is zero or invalid,
   return "ERROR INVALID REQUEST\n"
-- if the file does not exist, return "ERROR NO SUCH FILE\n"
-- if the file byte range is invalid, return "ERROR INVALID BYTE RANGE\n"
-- return an acknowledgement if successful; use the format below:

      ACK <bytes>\n<file-excerpt>


LIST\n
-- server returns a list of files currently stored on the server
-- the list must be sent in alphabetical order (case-sensitive)
-- the format of the message containing the list of files is as follows:

      <number-of-files> <filename1> <filename2> ... <filenameN>\n

-- therefore, if no files are stored, "0\n" is returned
```

For any other error messages, use a short human-readable description matching the format shown below. Expect clients to display these error descriptions to users.

```
      ERROR <error-description>\n
```

Note that all commands are case-sensitive, i.e., match the above specifications exactly. Make sure that invalid commands received by the server do not crash the server. In general, return an error and an error description if something is incorrect or goes wrong.

Be very careful to stick to the protocol or else your server might not work with all clients (and with all tests we use for grading).

Output Requirements

Your server is required to output one or more lines describing each command that it executes. Required output is illustrated in the example below. The `child` IDs shown in the example are either thread IDs or process IDs. And as per usual, output lines may be interleaved.

```
bash$ ./a.out 9876
Started server
Listening for TCP connections on port: 9876
Rcvd incoming TCP connection from: <client-hostname-or-IP>
[child 13455] Received PUT abc.txt 25842
[child 13455] Stored file "abc.txt" (25842 bytes)
[child 13455] Sent ACK
[child 13455] Received GET xyz.jpg 5555 2000
[child 13455] Sent ERROR NO SUCH FILE
[child 13455] Client disconnected
...

Rcvd incoming TCP connection from: <client-hostname-or-IP>
[child 11938] Received PUT def.txt 79112
[child 11938] Stored file "def.txt" (79112 bytes)
[child 11938] Sent ACK
[child 11938] Client disconnected
...

Rcvd incoming TCP connection from: <client-hostname-or-IP>
[child 19232] Received GET abc.txt 4090 5000
[child 19232] Sent ACK 5000
[child 19232] Sent 5000 bytes of "abc.txt" from offset 4090
[child 19232] Received LIST
[child 19232] Sent 2 abc.txt def.txt
[child 19232] Client disconnected
...
```

Submission Instructions

To submit your assignment (and also perform final testing of your code), please use Submitty, the homework submission server. The specific URL is on the course website.

Be sure you submit only your server code (i.e., do **not** submit any client code).

Note that this assignment will be available on Submitty a few days before the due date. Please do not ask on Piazza when Submitty will be available, as you should perform adequate testing on your own Ubuntu platform.

That said, to make sure that your program does execute properly everywhere, including Submitty, use the techniques below.

First, as discussed in class (on 9/7), output to standard output (`stdout`) is buffered. To ensure buffered output is properly flushed to a file for grading on Submitty, use `fflush()` after every set of `printf()` statements, as follows:

```
printf( ... );     /* print something out to stdout */
fflush( stdout );  /* make sure that the output is sent to a */
                   /*  redirected output file, if specified  */
```

Second, also discussed in class (on 8/31), use the `DEBUG_MODE` technique to make sure you do not submit any debugging code. Here is an example:

```
#ifdef DEBUG_MODE
    printf( "the value of x is %d\n", x );
    printf( "the value of q is %d\n", q );
    printf( "why is my program crashing here?!" );
    fflush( stdout );
#endif
```

And to compile this code in "debug" mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -D DEBUG_MODE homework4.c -pthread
```