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

Overview

- This homework is due by 11:59:59 PM on Thursday, December 1, 2016.
- This homework will count as 9% 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.
- Your program **must** successfully compile and run on Ubuntu v14.04.5 LTS or higher.
- Your program **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.

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 via TCP sockets to your server on a specific port number (i.e., the listener port); this port number is the first command-line argument to your server. Further, your server must **not** be a single-threaded iterative server. Instead, your server must use multiple threads or multiple child processes. 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.). Though you will only submit your server code, feel free to create one or more test clients. Test clients will **not** be provided. You should also use `telnet` and `netcat` to test your server.

Application-Level Protocol

The application-level 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. Clients connect to the server and can add and read files; clients can also request a list of files available on the server.

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** simply assume that files contain strings; in other words, do **not** use string functions that rely on the `'\0'` character.

Your server must use a connection-based protocol at the transport layer (i.e., TCP). 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.

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

```
STORE <filename> <bytes>\n<file-contents>
-- add file <filename> to 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


READ <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 "ACK" if successful, following it with the length and data, as follows:

      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
-- 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 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 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.

You should 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.

Further, subdirectories are not to be supported. A filename is simply a valid filename without any relative or absolute path specified. We will test using files containing only alphanumeric and `'.'` characters; you may assume that a filename starts with an alpha character.

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.

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

...

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

...

Received incoming connection from: <client-hostname-or-IP>
[child 19232] Received READ 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 URL is on the course website.

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