Project 5: Word-Count Network-Based Client-Server

Due Date: 5/9 by 11:59p

No Late Submissions or extensions.

Important Reminder: As per the course Academic Honesty Statement, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

The document first describes the aims of this project. It then gives the requirements as explicitly as possible. It hints at how the project may be implemented. It then briefly describes tools which may be used for this project. Finally, it lists exactly what needs to be submitted.

Aims

The aims of this project are as follows:

- To introduce you to network programming.
- To expose you to minimal aspects of multi-threading.
- To familiarize you with using the POSIX signal API.
- To give you some exposure to logging the operation of a daemon.

Requirements

Write a client program with executable `word-count` and server program with executable `word-countd`. Between them, the client and server should be able to compute word-counts as per your previous projects, with the server computing the actual word-counts and the client merely invoking the server and displaying the results.

The server should be started using the command:

```
$ ./word-countd DIR_NAME PORT
```

When started, the server should start up a daemon process running in directory `DIR_NAME` (which must exist). The initial server process should exit after printing the PID of the daemon process followed by a newline on standard output. The daemon process should become a TCP server listening for incoming network connections on port `PORT`.

It should be possible to terminate the daemon with a `SIGTERM` signal. On receipt of this signal, the daemon should clean up any allocated resources and exit cleanly.

The daemon should log its operation in `DIR_NAME/requests.log` for normal requests and `DIR_NAME/errors.log` for errors. These files should be created if they do not already exist; if they exist, then new entries should merely be appended to the previous entries.

Each log entry should consist of a single line containg `'|'`-separated fields. Each log entry should start with a ISO-8601 UTC timestamp in the format YYYY-MM-DDThh:mm:`ssZ`.

The errors log should have just one additional field containing a meaningful error message. Additionally, the errors log should contain log entries for starting/stopping the daemon.

In the requests log, the timestamp field should be followed by fields giving the individual arguments specified for the client starting with `N`, followed by a field containing the time (in milliseconds) needed to process the request.

The client should be run using the command:

```
   $ ./word-count HOSTNAME PORT N STOP_WORDS FILE1...
```

where `HOST_NAME` and `PORT` specifies the hostname and port where the daemon is running, and the remaining arguments are as in your previous projects. Note that all the files are assumed to be on the server; specifically, if `STOP_WORDS`, `FILE1...` specify relative paths, then they are assumed to be relative to `DIR_NAME` in which the server is running.

When invoked, the client should open a network connection to the server and send it the specified command-line arguments along with other necessary information. The server should process the request concurrently in a separate thread and return results to the client. The client should output on standard output the `N` most frequently occurring words in the one-or-more files `FILE1...` which are not in the file `STOP_WORDS`.

The functional specifications are identical to those of your previous projects:

- A **word** is a maximal sequence of characters for which the standard C library function `isalnum()` returns non-zero or is a single quote `'` (apostrophe `'\''`). So `is'nt` will be treated as a single word.
- Words which differ merely in case are regarded as identical.
- The output should consist of `N` lines with each line containing a lower-cased word followed by a single space character followed by the count of that word across all files `FILE1...`. The lines should be sorted in non-increasing order by count; ties where words have the same count should be broken with the lexicographically greater word **preceeding** the lexicographically smaller word.
- The program should handle files which do not necessarily end with newline.
- If the arguments to either the `word-countd` or the `word-count` programs are in error, then that program should print a suitable error message on standard error and terminate.
- Both programs should also detect any runtime errors (like memory allocation errors, I/O errors, process errors) and terminate after outputting a suitable error message on standard error. However, if a worker thread encounters an error, it should not stop the main daemon process; ideally, it should signal the error to the client so that it can be reported by the client.

The program must meet the following implementation restrictions:

- All communication between the client and server should be done using network communication.
- You may not assume any limits on the size of interactions between the server and client; i.e., you may not assume any limits on the the size of the command-line arguments to the client or the size of the response output by the client.
- Each client request should be handled by the server using a concurrent thread.
- If the server encounters an error, it should log that error in the error log. It should also make a best effort to report the error to the client. As far as possible, you should try to ensure that an error in a single request does not bring down the entire server.
- The client should exit only after cleaning up all resources (including any client-specific IPC facilities).
- The worker thread which handles each client request should clean up all resources before terminating.
- The server may assume that all distinct words in all of `FILE1...` and their counts can fit within memory.
- There should not be any implementation restrictions on the size of entities except those defined by available resources. Hence there should not be any restriction on the size of a word or a line.

Caution

In this project, you will be creating processes and daemons. It is possible that a buggy program may create more processes than you desire. To clean up processes first use the `ps` command which will list out all your processes. Identify the processes for your project by the `CMD` field and note their PID. Then kill them using `kill -9` *PID1 PID2 ...* .

Please make sure you terminate any daemon processes you may have running whenever you logout.

Port Assignments

You will need a port number to test your server. To avoid students stepping on each other's ports when working on the `remote.cs` machines, please complete the web form at http://zdu.binghamton.edu/cs551s16/misc/ports/ to get a range of 5 ports you should use. You will need to enter your email address and then click on the link in the email you should receive. You should only use ports in the range assigned by this form.

NOTE: This form will only work on the site http://zdu.binghamton.edu/cs551s16. Specifically, it will not work on the mirror'd web site at `cs.binghamton.edu`.

Note that this script only ensures that other students in this course will not be using the ports assigned to you. It can be the case that other applications may be using some of those ports. In that case, do not use those ports or obtain a different set of ports from the port assignment form by running it again using a different email address.

[All the relevant code for the form is available under the course `misc/ports` directory.]

Hints

Some things to watch out for during your implementation:

- Since your server is multi-threaded with concurrent requests handled using concurrent threads, you need to ensure that your code for the word-count is free of any unsafe uses of global variables.

- If your server terminates unexpectedly, you may find that you are unable to immediately restart the server on the same port because of the TCP TIME-WAIT state. You could try using another port number (within your range of 5 assigned ports), but a better fix is to setup the server to allow reusing of the same port. This can be achieved by something like the following `setsockopt()` call:

  ```
  /* socket created with returned socket descriptor s; before bind().
   */
  const int on = 1;
  if (setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on))) {
    ... error code ...
  }
  ```

- The campus network blocks incoming requests to non-standard ports. Hence if you are running on `remote`, please use a browser on `remote` or within campus. You will not be able to use a off-campus browser to access your server running on `remote` unless you setup a `ssh` tunnel from your local machine to the `remote` instance and browse the forwarded port on your local machine.

The following steps are not prescriptive:

1. Review material on multi-threading, network programming, signals and logging.

2. Decide exactly how you will handle the project requirements. The networking and threading requirements are relatively simple to implement. The implementation for the logging requirement may be slightly more involved. The termination requirement will probably require the most thought.

3. Design the protocol between server and client. It is strongly suggested that you use a text protocol.

4. Pull together a solutions to one of the previous projects; ensure that it does not have any unsafe global state. Set it up as an application with the `main()` program being minimal. Test it to make sure it works ok.

5. Add code to your application from the previous step to change it into a network daemon. This should simply involve reusing code given in the slides, text or widely available on the web. At this point you should have a simple iterative server. Assuming that you are using a textual protocol, you should be able to test your server using `nc` or `telnet`.

6. Build the `word-count` client. Test it with your server.

7. Add logging code to your server to meet the logging requirements. You may find the `time()`, `strftime()` `clock_gettime()` POSIX routines useful.

8. Add code to your server to make it a concurrent server, with concurrent requests being handled using concurrent threads.

9. Add signal handling code to allow stopping the server cleanly using `SIGTERM`.

10. Test to ensure that you meet all requirements. You may want to use a tool like valgrind to ensure that both your client and server clean up all resources.

11. Ensure that you deactivate any debugging statements you have in your code before submitting. Hence the server should be completely silent and the client should only omit the required output.

Submission

You will need to submit a compressed archive file `prj5.tar.gz`. This archive must contains **all** the **source** files needed to build your project; specifically, when unpacked into a directory followed by the command `make` within that directory, your `word-count` and `word-countd` executables must be built.

Additionally, this archive **must** contain a `README` file which should minimally contain your name, email, the status of your project and any other information you believe is relevant.

Note that it is your responsibility to ensure that your submission is complete so that simply typing `make` builds both executables. To test whether your archive is complete, simply unpack it into a empty directory and see if it builds and runs correctly.

Submit your project using the submission link for this project, under **Projects** in Blackboard for this course.