Project 3: Word-Count Client-Server

Due Date: 3/25 by 11:59p

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 building Unix daemons.
- To expose you to IPC using FIFOs.
- To familiarize you with the Unix dynamic loading API.

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
```

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 should listen for a request from a client on a well-known FIFO in directory `DIR_NAME`, creating the FIFO if it does not already exist. When a request is received, it should process it and return the results to the client.

The client should be run using the command:

```
$ ./word-count DIR_NAME WORD_CHAR_MODULE N STOP_WORDS FILE1...
```

where `DIR_NAME` specifies the directory for the server, `WORD_CHAR_MODULE` specifies the name of a dynamically-loaded module which defines a function `isWordChar()`, and the remaining arguments are as in your previous projects. If `WORD_CHAR_MODULE`, `STOP_WORDS`, `FILE1...` specify relative paths, then they are assumed to be relative to `DIR_NAME`.

When invoked, the client should send a request to the server using the well-known FIFO in `DIR_NAME` sending it the command-line arguments starting with `WORD_CHAR_MODULE`. The server should process the request concurrently in a separate worker process and return results to the client. The client should output on standard output the `N` most frequently occurring words in one-or-more files `FILE1...` which are not in the file `STOP_WORDS`.

- A **word** is a maximal sequence of characters `c` for which the function `isWordChar(c)` in `WORD_CHAR_MODULE` returns non-zero.
- 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 process 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 named pipes (FIFOs). (The details of the exact communication protocol are up to you)
- Each client request should be handled by the server using a concurrent worker process spawned using the double-fork technique to avoid zombies.
- The client should exit only after cleaning up all resources (including any client-specific FIFO's).
- The worker process which handles each client request should clean up all resources before terminating.
- The programs may assume that the the client command-line arguments starting with `WORD_CHAR_MODULE` plus a PID can be contained within `PIPE_BUF` bytes using any reasonable scheme.
- 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.

Also, be aware of the fact that FIFO communication is local to a machine which can be a problem since `remote.cs.binghamton.edu` is a cluster of machines; the machine you are currently logged on to may be different from the machine you previously logged on to.

Provided Files

You are being provided with the following:

README
: A template README; replace the `XXX` with your name, B-number and email. You may add any other information you believe is relevant to your project submission.

word-char.h
: The specification file for dynamically loaded modules.

alnum-quote.c
: A module which defines a word-char as a alphanumeric or single-quote char.

non-space.c
: A module which defines a word-char as a char which is not a space.

modules-test.c
: A test program for the above modules. Uses the dlopen API to load the module specified by the first argument and splits the remaining arguments into words as per the definition of a word-char specified in the module.

Makefile
: This file can be used to build your project. You will need to edit it to suit your project organization; minimally, you will need to list the source files constituting your project in by completing the definition of `PROG_FILES` and also list the object files needed for your client and server by completing the definitions of `CLIENT_OBJS` and `SERVER_OBJS`.

  The `Makefile` provides the following targets:

  all
  : This default target should build both your client and server as well as the provided dynamically loaded modules.

modules-test
: Builds a test program for the dynamically loaded modules.

DEPEND
: Produces dependency information for all the `.c` files in the current directory. This information can be cut-and-pasted onto the end of the `Makefile`.

The Library Path

You will need to ensure that your `LD_LIBRARY_PATH` contains the path to your dynamically loaded `WORD_CHAR_MODULE` modules. Assuming that they are in the current directory, ensure that your `LD_LIBRARY_PATH` contains `.`. As usual, the setting will use different syntax depending on whether you are using a `sh` or `csh` type shell.

Testing the Modules

The following log shows a test of the modules:

```
$ make modules-test
gcc -g -Wall -std=c11 -fPIC -shared alnum-quote.c -o alnum-quote.mod
gcc -g -Wall -std=c11 -fPIC -shared non-space.c -o non-space.mod
gcc -g -Wall -std=c11 -fPIC modules-test.c -ldl -o modules-test
$ LD_LIBRARY_PATH=. ./modules-test alnum-quote.mod "ab’c #!@$%^ x123’"
ab’c
x123’
$ LD_LIBRARY_PATH=. ./modules-test non-space.mod ’#!@$%^ a;b"d x123’
#!@$%^
a;b"d
x123
$
```

Hints

The following steps are not prescriptive:

1. Review the material on daemons, FIFOs and dynamically loaded modules. Design a protocol to communicate between your client and server. The protocol should also take care of the creation and cleanup of any client-specific FIFOs.

2. Start your client and server code using your code for prj1 or the solution.

3. Remove the hard-coded definition for what constitutes a word-char. Instead, add a additional argument to the program specifying the module and change the program to load the module and use the word-char definition provided by the module. Test.

4. Modify the program to become a daemon meeting all the specifications for `word-countd` except that instead of writing the results back to the client; instead have the worker process simply write the results to standard output (assuming you have not closed it). You can test by simply `echo`’ing a client request to your well-known FIFO.

5. Write a simple client program which when run, simply writes its arguments (starting with `WORD_CHAR_MODULE`) to the well-known FIFO. Test.

6. Modify your server to return the results to the client according to your protocol and modify the client to write those results onto standard output as per the specifications. Test.

7. Iterate until your client and server meet all the specifications.

Submission

You will need to submit a compressed archive file `prj3.tar.gz` which contains all the files necessary to build your `word-count` and `word-countd` executables. 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 (it is probably a good idea to mention which data-structure you have chosen for your word-store).

If you have edited the provided Makefile to indicate your project files, then you can use its `submit` target which will build the compressed archive for you; simply type `make submit`.

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.