Project 4: Word-Count Client-Server with POSIX IPC

Due Date: 4/18 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 POSIX IPC.
- To expose you to use file locking to exclude multiple daemon instances.
- To familiarize you with writing Makefiles.

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 process should try to get an exclusive lock on the text file `.pid` in `DIR_NAME`; if unsuccessful it should exit with failure after printing an error message on standard error containing the PID of the process currently holding the lock; if successful, it should set the contents of `.pid` to its PID (followed by a newline).

The daemon should listen for requests from clients on a well-known shared memory segment, creating the segment if it does not already exist.

The client should be run using the command:

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

where `DIR_NAME` specifies the directory for the server, and the remaining arguments are as in your previous projects. If `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 shared memory segment, sending it the specified command-line arguments along with other necessary information. 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 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 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 possibly multiple shared memory segments implemented via any of the techniques discussed in class. The size of each individual memory segment should not exceed 4 KiB.
- You may assume that the client's command-line arguments plus any reasonable overhead can fit within a single memory segment.
- You may **not** assume that the response to a single client request can fit within a single memory segment.
- You should use any of the variants of POSIX semaphores discussed in class as your only synchronization mechanism.

- POSIX IPC object names live in a global namespace. Hence to prevent clashes of POSIX names, all the POSIX names used by your project **must** start with the following prefix: a `/` character, followed by the login-name of the user running the client or server.
- 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 IPC facilities).
- The worker process 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.

You can list out all POSIX objects on the system by doing a `ls` listing of the `/dev/shm` directory. You can then do a normal file remove using `rm` to remove a POSIX object. Please make sure to clean up all IPC objects belonging to you when you log out.

Hints

The following steps are not prescriptive:

1. Review the course material on shared memory and POSIX IPC.
2. Design a protocol for communication between server and client. Make sure that your protocol can handle responses of arbitrary size while using shared memory segments of size not exceeding 4 KiB.

   Designing this protocol while coding can be a time-sink; hence, it is imperative that you design this protocol before you start coding.
3. Since you are not being provided with a `Makefile` for this project, you will need to create your own. Start with a `Makefile` from one of your earlier projects and modify it as you go along. To avoid mysterious `make` errors, ensure that all shell commands in the Makefile start with a leading tab and lines being continued are terminated with a trailing backslash (without any following linear

space). Make sure to specify all file dependencies so as to ensure that targets get remade whenever their prerequisites change.

4. A solution to prj1 or prj3 can serve as a useful starting point for this project. The former has the advantage that it does not contain any code not needed for this project like code for FIFO-based IPC or dynamic loading of modules. The latter has the advantage that it already has the code for creating a daemon and worker process; what you would need to change is the IPC from self-synchronized FIFO's to semaphore-synchronized shared memory.

5. Adapt the exclusive file locking from the class notes to ensure that multiple instances of the daemon cannot run in the same directory `DIR_NAME`.

6. Iterate the above steps until you satisfy all requirements.

Submission

You will need to submit a compressed archive file `prj4.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.