CSE-376 (Spring 2021) Homework Assignment #4

Version 2 (2021-04-18)

Due Tuesday, May 4, 2021 @ 7:00 pm EST (19:00 ET)

This assignment is worth 25-30% of your grade, prorated to all assignments.

1. Purpose

Put together much of the knowledge gained in this course, design, develop, and test code to demonstrate your low-level understanding of low-level Unix system calls. You will develop a client/server system for job submissions. You are free to use just about any system call, library function, etc. that's on the two test machines, whether we described it in class or not. Hopefully by now you're able to find "new" system calls to use, study their man pages, test how they work, and use them in your code.

2. Demos

To expedite grading, and also allow you to show off your work, we'll have short demos (30-45 min) per student. Demos will be held over a period of 3-5 days following the due date. We will use Zoom and record the demos too. More info TBA (e.g., demo signup sheet, procedures).

3. Task

This is a more open ended assignment, on purpose: consider it a "class project".

The goal is to create a "job submission" system: a client that can submit jobs to be executed with various criteria; and a server that will process these jobs asynchronously and report their status. We leave the design details up to you, but here are the minimum requirements (meaning you can add additional features as you see fit, and we'll consider them as extra credits).

3.1. Server

The server will have a number of configuration options that you can pass on the command line:

1. The maximum number of jobs that can be concurrently run. This is to ensure we don't blow up the system.
2. The server will track resources used by each running job (a process): time and CPU cycles consumed, memory used, etc. The server will be able to take action on a process that exceeds these caps: kill a job, stop it, use nice(2) to change it to a different priority, etc.

3. The server will be able to report back to the client the status of one or more jobs: list one or more jobs, resources they're using, are they running or not (and why not), and the output these jobs produced on stdout/stderr.

3.2. Client

The client will be able to issue commands to the server to list one or more jobs, check their status, tell the server to suspend/kill a process or change its "nice" priority, retrieve status/error codes, etc.

The client could submit new jobs. New jobs will be described as a vector of argv[] and envp[] strings, along with an additional data structure that describes the restrictions on the job to run (max time to run, max resources to consume, priority, etc.).

3.3. Client/Server Interaction

The client and server should communicate via a Unix named pipe (FIFO). This is a special file you create on the file system that has two communication channels. The server will create the pipe and listen for commands on it (e.g., "/home/jdoe/.hw4server_control"). The client could write(2) to the pipe and can read(2) back from it. When one side writes to a pipe, the other side can read back from it. Otherwise a reader trying to read from a pipe would be blocked waiting for the other side to write something. (Consider using poll(2) or select(2) like calls for the server.)

You will have to design a small *protocol* for communicating over the pipe. A good protocol starts with a short code that determines the type of operation to perform and the type of response that is returned. For example, the first byte can be 1 to mean "submit a new job", 2 to mean "list existing jobs", 3 to mean "kill an existing job", etc. Figure out how many commands you need and give them some unique command ID. For example, the client could submit the kill to an existing job: first byte has the value 3; next byte lists the job ID to kill. Or, for a more complex operation like submit a new job:

- first byte is 1 (command type number)
- next 4 bytes (an int) say how many bytes are in the whole message, to help the server verify the message size and malloc(3) enough space to hold it.
- next byte is the same as argc
- then come all bytes of argv, delimited by \0's
- then come envp[] array (you may prefix this array with a count of number of envp elements, ala argc)
- then comes a C struct that denotes limits: first value could be "max time", second could be "max memory", 3rd byte could be "priority value".

You then write(2) that data to the pipe. And if you've set things right on the server, the server will wake up and be able to read from the pipe: it can read one byte to determine the command type, and then read the rest incrementally or in one shot.

The client can then go into a read(2) loop waiting for a response: when the server, say, writes a return status reply, the client will be able to read and process/display it. For example, the server can write a reply message that looks like this:

- first byte is 1 (meaning a REPLY to a "submit job command")
- next byte can be an int stating whether the submitted job was received successfully (0) or failed (errno). Note this is NOT an indicator of the exit status of a job submitted.
- last byte would be optional: if the command succeeded, you can return a unique job ID assigned to this command.

Note that you will need a different client/server command (or two) to return the stderr and/or stdout output of a command that finished or was terminated. It would help if each submitted job would have a unique job ID, so you can send a request to "query status of job N".

For this assignment, assume all submitted programs are non-interactive: that is, they do not read from stdin. This will simplify the assignment. But also, in most job-submission systems, the work submitted is self-contained and does not depend on user interaction (e.g., submitting a weather simulation to an HPC cluster).

3.4. Design Document

Please write a design document describing the requirements, design, and implementation of your system: you must describe your protocol and, client submission tool flags/options, key data structures, and system operation. In other words, make this design document sufficiently detailed that anyone reading it could re-implement your system from scratch. Use 11pt TimesRoman font, 1” margins, and number each page. Ensure your name is listed at the start of the document. I expect the document to be at least 2 pages but no more than 6 pages. Feel free to include code snippets if it helps, figures you draw, etc. The design document will be worth at least 10 points (out of 100) in this assignment. You must submit your design doc as a file named “design.pdf” (all lower-case, PDF format only); if you author this design.pdf doc with something else, it is ok to also include the source file (latex, docx, etc.).

4. Test Scripts

You know the drill. Write and include as many or as few test scripts/programs to show us that your features work. The easier it’d be for us to test your code, the happier the grader(s) will be ...

5. Style and More

Aside from testing the proper functionality of your code, we will also evaluate the quality of your code. Be sure to use a consistent style, well documented, and break your code into separate functions and/or source files as it makes sense.

To be sure your code is very clean, it must compile with "gcc -Wall -Werror" without any errors or warnings! If the various sources you use require common definitions, then do not duplicate the definitions. Make use of C's code-sharing facilities.

Because in this assignment you will include a design document (see above), there’s no need to include a separate README file. Your design document should include what we normally ask in a README: describe what you did, what approach you took, results of any measurements you made, which files are included in your

submission and what they are for, etc. Feel free to include any other information you think is helpful to us; it can only help your grade. The code you write should be your own, but if you want to use any online code, you must clear it with me (note: github sources NOT allowed), and cite it both in your code and your design document.

6. Submission

You will need to submit all of your sources, headers, scripts, Makefiles, and README. Submission is accepted via GIT only! Do not submit regenerable files like binaries, *.o files, or any temp files — only true "source" files.

PLEASE test your submission before submitting it, by checking it out in a separate directory, compiling it cleanly, and testing it again. DO NOT make the common mistake of writing code until the very last minute, and then trying to figure out how to use GIT and skipping the testing of what you submitted. You will lose valuable points if you do not get to submit on time or if your submission is incomplete!

7. Extra Credit

If you do any of the extra credit work, then your EC code must be wrapped in the following macro exactly. Please do not use any other ifdef other than EXTRA_CREDIT precisely as spelled; deviations will result in point deductions.

```
#ifdef EXTRA_CREDIT
      // EC code here
#else
      // base assignment code here
#endif
```

This extra credit is worth up to 50 extra points (the main assignment is worth 100 points).

7.1. Network TCP socket (10 points max)

Instead of using a localhost Unix pipe, use a network TCP socket (frankly this should be easy once you get Unix pipes working). Study the networking system calls like socket(2), connect(2), listen(2), etc. Since we have two test machines, run the server on one machine and the client on another.

Your server will need to have an option to decide which port number to listen on, and the client will have to connect to that port number. Unix port numbers can go as high as 65,636 (ports 1-1024 are “reserved” for root only). To avoid port conflicts (trying to connect to another student’s server), I suggest you use port numbers above 10,000 where the last 4 digits match the last 4 digits of your SBID number (which are unique for this class). For example, if your SBID is 123456789, then use port numbers 16789, 26789, 36789, 46789, or 56789.

7.2. Feed in stdin remotely (10 points max)

Add an option so the client can take an input file and feed it (by streaming it) to a specific job on the server. The job would normally be blocked waiting on stdin input, but once the streamed data arrives to the server-side job, the job should unblock, process the incoming data as if it read it from stdin, and when the stream ends, close stdin (like a ^D given on the command line). Note that this stdin data feed should be started after the job was already submitted and began running.

7.3. Open ended extra features (20 points max)

Because this is a more open-ended assignment, there are no explicit extra credit features. However, feel free to add additional features as you feel it makes sense. You can ask me privately or publicly (e.g., piazza) and I'll let you know if I think such features make sense (please don't ask me how much this or that extra feature would be worth but you can roughly tell by the scope of other EC for this assignment). Think about useful features to someone using such a system, efficiency, and more. Impress us, document all your features, and demo them.

7.4. Early submission (10 points max)

To incentivize you to submit your code earlier than the deadline, we'll give you 2 points if you submit your assignment at least 24 hours before the official deadline; and 2 more points for every 24 hours earlier you submit. Note that we count the LAST git-push to your repo as the last time you submitted the assignment, even if you made a very small change.

Good luck.

8. Copyright Statement

9. ChangeLog

- V1: initial draft
- V2: TA review