April 6th, 2022

CSE344 – System Programming – Midterm project - v1

Context

Servers consist commonly of very powerful hardware in terms of both processing and storage capacity, so as to be able to handle the high number of often concurrent client requests. However, the centralization of such tasks can lead to financial disadvantages (only some companies can afford huge servers to handle the incoming traffic).

An inefficient alternative approach, is to organize multiple mid- or even low-end machines serially. For example, say client X connects to server Y. If server Y is not busy, then it processes the request of X and responds to X. Otherwise, if Y is busy handling other clients, then Y forwards X's request to another server Z who processes it and responds to client X.

Objective

The main idea of this project is to simulate the aforementioned paradigm with 2 process-pooled servers: Y and Z executing on the same system as the clients.

There are 3 programs to be implemented: client X, server Y and server Z. There will be **a single instance of Y and of Z**, and there can be an **arbitrary number of client processes X**, all running concurrently. Each client will submit a matrix to the server Y, and receive a response (from Y or Z) about whether it is invertible or not.

0) Data

The data file will be in CSV (ASCII) format. It will be comma separated with no quotation marks. It will contain n integer values in each of its first n rows (n >= 2). The file contents will denote a **square real** matrix of size n.

1) Client X

```
./client -s pathToServerFifo -o pathToDataFile
```

where
-s: the path (relative or absolute) of the server fifo
-o: the path (relative or absolute) of the data file to be processed by the server.

The client will connect to the server Y through the server fifo given as a commandline argument, send its request, receive its response through its client fifo, print it to STDOUT and exit. In case of SIGINT it will exit gracefully (i.e. printing an informative message, closing open files, removing temporary files and freeing all allocated resources).

The format of the request is up to you; be careful, the matrix has an arbitrary size.

Client output (**every row must begin with a timestamp**):

```
Client PID#667 (../../demo/22/exam/data.csv) is submitting a 5x5 matrix
Client PID#667: the matrix is invertible, total time 2.25 seconds, goodbye.
```

2) Server Y

Server Y and Z will be daemon processes; this means that you should:

- take measures against double instantiation
- make sure they have no controlling terminal (as explained in our lectures)

Server Y is tasked with receiving requests from clients X, processing them, and then responding to the clients, until it receives a termination signal SIGINT.

```
./serverY -s pathToServerFifo -o pathToLogFile –p poolSize -r poolSize2 -t 2
```

where
-s: denotes the path (relative or absolute) of the server fifo
-o: denotes the path (relative or absolute) of the log file to record all server output/errors
-p: denotes the pool size of serverY (p >= 2)
-r: denotes the pool size of serverZ (r >= 2)
-t: denotes the sleep duration (in seconds)

Server Y is a process-pooled process. This means that it will have p children processes (i.e. workers). They are there in order to actually handle the arriving requests, so that server Y will be free in the meantime to receive additional requests, thus increasing the server's efficiency.

Once a request arrives through the serverFifo, it will send it (using a pipe) to any available worker process. **But how will it know which worker is available? Design and implement a solution.**

If a worker from the pool is available, the request will be sent to it through its pipe. The worker will handle it: meaning that it will read the client's request, compute whether the matrix is invertible (with a method of your choice, I recommend the cofactor method – no libraries, you are expected to implement it), and respond directly to the client either yes or no. The pool workers will simulate intensive processing by sleeping for t seconds. Afterwards, the worker will wait for the next request (**how?**).

However, if all worker processes of serverY are already busy handling other requests, then serverY will forward the incoming request to serverZ (using a pipe).

In case of SIGINT, serverY, serverZ and all of their workers will exit gracefully.

Y output (all output and errors are to be written to the log file):

```
Server Y (../../demo/22/exam/y.log, p=4, t=2) started
Instantiated server Z
Worker PID#777 is handling client PID#667, matrix size 5x5, pool busy 1/4
Worker PID#778 is handling client PID#669, matrix size 3x3, pool busy 2/4
...
Worker PID#771 is handling client PID#662, matrix size 3x3, pool busy 4/4
Forwarding request of client PID#700 to serverZ, matrix size 3x3, pool busy 4/4
...
Worker PID#778 responding to client PID#669: the matrix IS NOT invertible.
...
SIGINT received, terminating Z and exiting server Y. Total requests handled: 16, 9
invertible, 7 not. 6 requests were forwarded.
```

Every line of output by Y and Z must begin with a timestamp (i.e.: date & time down to seconds).

3) Server Z

Server Y will instantiate server Z as a separate process; server Z will not be executed explicitly by the user. It will wait for forwarded requests from serverY, handle them and write its output to the same log file as Y (**use file locks to synchronize access**) and run indefinitely until terminated by Y.

Server Z will have only one process pool of size `poolSize2`,and will delegate to them whatever request arrives through its pipe. It will forward them to an available worker, and each worker will handle them in the same way as in serverY.

The difference is that this time serverZ will communicate with its workers via a shared memory segment and NOT through pipes. The serverZ will place the request on a queue located at the shared memory segment, and any worker that becomes available will handle it. If there are not pending requests the workers will wait until there is one. Take precautions against the synchronization problems while accessing the shared memory segment.

Z output:

```
Z:Server Z (../../demo/21/exam/y.log, t=2, r=4) started
Z:Worker PID#800 is handling client PID#700, matrix size 5x5, pool busy 1/4
...
Z:Worker PID#800 responding to client PID#700: the matrix IS NOT invertible.
...
Z:SIGINT received, exiting server Z. Total requests handled 6, 3 invertible, 3 not.
```

Evaluation

Your project will be evaluated at the department lab (running ubuntu) with an arbitrary number of clients and commandline parameters. **You must participate at the demo session for a valid grade**. It will be tested with varying numbers of clients and varying matrix sizes. I'm expecting to see processes communicating perfectly, well-synchronized, and producing correct results. You can use any of (and only) the methods covered in our lectures until and including week 7.

Rules

1) Compilation error: grade set to 1; if the error is resolved during the demo, then evaluation continues.
2) Compilation warning (with respect to the **-Wall** flag); -1 for every warning until 10. -20 points if there are more than 10 warnings; no chance of correction at demo.
3) No makefile: -30
4) No pdf report submitted (or submitted but insufficient, e.g. 3 lines of text with no design explanation, etc): -20. Other file formats are not admissible.
5) If the required command line arguments are missing/invalid, your program must print usage information and exit. Otherwise: -10
6) The program crashes or doesn't produce expected output with normal input: FAIL
7) The program doesn't satisfy a requirement: FAIL
8) Presence of memory leak (regardless of amount – checked with valgrind) -30
9) Late submissions will not be accepted
10) In case of an arbitrary error, exit by printing to stderr via perror a nicely formatted informative message. Otherwise: -10
11) Clean-up after the children processes, shared memory segments; no zombie processes, -50 otherwise.
12) Take precautions against signals interrupting your operations. -10 otherwise.

What to submit:

- Your source files, your makefile and a report; place them all in a directory with your student number as its name, and zip the directory.
- Your report must contain: how you solved this problem, your design decisions, which requirements you achieved and which you have failed.
- The report must be in English.
- Your makefile must only compile the program, not run it!
- Do not submit any binary executable files. The TAs will compile them on their own boxes.
- Your code will be compared against online sources; you are free to be inspired, but you are also expected to write your own code.
- Proven cases of plagiarism will be punished to the full extent.

Good luck