Project 2 Multi-Process Matrix Multiplication

Due Date: 3/3 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.

Aims of This Project

The aims of this project are as follows:

- To expose you to creating multiple processes in Unix.
- To use anonymous pipes for inter-process communication.
- To provide you with experience designing IPC structures and protocols.

Project Specification

Provide an implementation of the functions specified in ./files/matrix_mul.h.

When your implementation is linked with the provided ./files/main.c main program, the resulting executable program should be able to perform matrix multiplication. Specifically, the executable will be run by a client process which calls your implemented functions to perform matrix multiplications. All the dot product computation required for matrix multiplication **must** be performed only by worker processes started directly or indirectly by the client process. All IPC is restricted to anonymous pipes.

Your implemented functions should check for most errors and try to ensure that errors are reported to their callers.

Provided Files

The ./files directory contains the following:

Makefile
: This makefile provides the following targets:

  `prj2`
  : This will build the `prj2` executable.

  `clean`
  : This will clean out all generated files.

  `submit`
  : This will build the required `prj2.tar.gz` archive.

Simply typing `make` will build the `prj2` program, typing `make clean` will remove all generated files and typing `make submit` will create a `prj2.tar.gz` compressed archive which can be submitted.

You may edit this file if you choose to use a different organization for your project. When editing, watch out for tabs (the first character of any command-line **must be a tab character**).

README
A template README; replace the `XXX` with your name, B-number and email. It should give a brief high-level overview of your IPC structure and protocol. You may add any other information you believe is relevant to your project submission.

./files/matrix_mul.h
This header files contains the specifications for the functions you need to implement. You should **not** change this file.

./files/matrix_mul.c
This file contains skeletons for the functions you need to implement. You will need to add your code to this file. You may choose to use additional auxiliary files (in which case you will need to adjust the `Makefile` suitably).

LOG
A log of compiling and running this project.

main.c
A main program which allows you to run tests involving both random data and data read from external files. The command-line usage implemented by this program is shown in the above `LOG` file. A particular option (`-g` or `--gold`) allows you to use a simple in-process matrix multiplication instead of the multi-process matrix multiplier you are requested to implement.

You should **not** change this file.

matrix.dat
This file contains some test matrices. A test matrix is represented by the following sequence of whitespace separated entries:

- A description (which cannot contain whitespace).
- A `int` giving the number of rows `nRows`.
- A `int` giving the number of cols `nCols`.
- The matrix entries: `nRows` x `nCols` whitespace-separated numbers listed in row-major format.

./files/matrix_test_data.c, ./files/matrix_test_data.h
A module used to implement the reading of test matrices. You should not need to change these files.

Note that you can build a working executable program by copying these files into an empty directory and typing `make`. You can run matrix multiplications using the *gold* matrix multiplication routine. In fact, you can even run non-gold matrix multiplications using the empty skeleton functions in `matrix_mul.c`; the tests will simply fail since the result matrices will contain garbage.

Caution

You will be creating multiple processes in this project. Keep the number of processes created under 10 and be careful with your code to ensure that you do not spawn an exponential number of processes. It is trivially easy to create *fork bombs* purposely or inadvertently (consider omitting the `break` statements within the loops in the process chain/fan discussed in class).

You can use `ps` to list out your processes (do `man ps` for the manual). You can kill a process with pid `PID` using `kill -9 PID` (`9` is the usual signal number for the `SIGKILL` signal which is guaranteed to terminate the process if you have authorization).

The cs551 Library

The provided `main` driver uses some memory allocation and error reporting routines from the cs551 library. You should not need to use this library in your code since the matrix API reports all errors via error codes. However, you are welcome to use this library if necessary; the specs are in ../../include, the source code in ../../src/libcs551 and the precompiled library in ../../lib.

The provided Makefile is setup to link with this library.

Since the library is dynamic, the linking happens when (and after) you start your program. Hence when the program starts up, you need to let the system know where it can find this library. This can be done by setting the `LD_LIBRARY_PATH` environmental variable to contain the directory containing the library. This should already be setup in your environment.

Please use the command `echo $LD_LIBRARY_PATH` to check whether this environmental variable is setup properly for you. If not, please use the appropriate command below before running your program.

Under a `sh`-derived shell like `ksh` or `bash`, you would use

```
$ export LD_LIBRARY_PATH=$HOME/cs551-17s/lib:$LD_LIBRARY_PATH
```

whereas with a `csh`-based shell like `tcsh`, the syntax you would use is:

```
% setenv LD_LIBRARY_PATH $HOME/cs551-17s/lib:$LD_LIBRARY_PATH
```

Hints

You may choose to follow the following hints (they are not by any means required). They assume that you are using the project structure supported by the provided Makefile,.

For debugging you can use gdb or a gui frontend ddd. A cruder option is to simply add `fprintf(stderr, ...)` statements to your code; a slightly better option is to use the TRACE macro provided in the cs551 library.

1. Review material on C stdio, Unix I/O, processes and pipes. Make sure you understand the interaction between stdio buffering and Unix I/O as well the necessity for closing pipe write ends in order to see EOF on the read end.

2. Decide on the format used for data-interchange between processes. The main choices are:

   Text
   Numbers are represented using a textual representation. Your data stream could basically be a stream of whitespace separated numbers; errors could be indicated by using a non-number character followed by the error code.

   Binary
   Numbers are represented using their internal binary representation. The data stream could consist of a stream of fixed size structures where each structure could contain data as well as an error code.

   One advantage of binary representations is better efficiency. A major advantage of textual representation is easier debugging because the data stream is easily decipherable (this is more applicable in network protocols than with pipes).

3. Decide whether you would prefer to use Unix I/O or C stdio. You can move between these two layers using `fdopen(3)` and `fileno(3)`. One advantage of C stdio is buffering but that is also a disadvantage in that it can cause problems if you do not understand how it interacts with Unix I/O and multiple processes.

   This choice could be affected by your choice in (1). If you chose text, then choosing C stdio in this step may be preferable though not absolutely necessary (use the `printf(3)`, `scanf(3)` family of functions for number I/O). OTOH, if you chose binary, you could go with either Unix I/O (use plain-vanilla `read(2)` and `write(2)`) or C stdio (use `fread(3)`, `fwrite(3)`).

4. Decide on the topology of how you will connect your processes using pipes and the exact protocol you will orchestrate to perform a matrix multiplication. Some of the questions you will need to answer:

   - How many pipes will your implementation require and how will they connect the different processes?

   - How will you distribute the input data for each multiplication among the worker processes in order to maximize potential concurrency?

   - How will you send the input data from the client process to the worker processes? How will the worker processes send the dot-product results back to the client process? How will the client multiplication function know that the multiplication is complete so that it can return to its caller?

- How will you ensure that all worker processes will be cleaned up when `freeMatrixMul()` returns? Note that depending on your protocol, this can be done without using the `wait(2)` family of calls (though their use is permitted).

- How will you handle errors occurring in any of the 3 functions? How will errors occurring in the worker processes be conveyed back to the client process? How will cleanup work after an error?

5. You will save yourself a lot of time if you ensure that your design is solid before you start coding. So iterate steps 1 - 4 until you are satisfied with your design. You may want to test your design by simulating it on paper for doing something like multiplying a 4 x 4 matrix using 3 worker processes.

6. It may be a good idea to write little play programs experimenting with pipes and processes.

7. Implement your design. Incrementally test as you code. Some alternatives to see what is going on:

   - Use a debugger like `gdb` or `ddd`.

   - Use the `TRACE()` macro from the `cs551` library.

   - Use `fprintf(stderr, ...)` statements.

   You could start out testing your code by multiplying two 1 x 1 matrices. Make sure you test your code with matrices which are not evenly partitioned across the worker processes.

8. Memory leaks can be a major problem with C programs. This can be avoided by getting into the habit of using memory debugging tools. One such tool is valgrind; you can run it by simply preceeding the normal command-line used to run your program with the command `valgrind`. If the command reports problems, then it suggests the options you can use to debug the problems further. It will even help debug problems in the worker processes.

9. Test and review your code until it meets all requirements.

Submission

You will need to submit a compressed archive file `prj2.tar.gz` which contains all the files necessary to build your `word-count` executable. Additionally, this archive **must** contain a `README` file which should minimally contain your name, email, a brief description of your protocol, the status of your project and any other information you believe is relevant.

If you are using the suggested project structure, then the provided Makefile provides a `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 the `prj2` executable. 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.