Concurrent and Multiprocessing	**S21 CSCI460 Operating Systems**
Assigned: **2021-02-22**	Due Date: **See Parts**
Abstract:	Solve a matrix multiplication problem using processes and threads and various IPC mechanisms.
Objectives:	1. Process Control Blocks 2. Threads within a PCB 3. Interprocess Communication via Shared Memory 4. Communication between threads via Global Memory Objects 5. Using C Pointers to address matrices in contiguous memory
Grading: 45 pts	A (≥41.85); A- (≥40.50); B+ (≥39.15); B (≥37.35); B- (≥36.00) C+ (≥34.65); C (≥32.85); C- (≥31.75) D+ (≥30.15); D (≥28.35); D- (≥27)
Outcomes:	R8 (CS: 5-1, 2, 5-a-4; SE: 1, III-1-2-1) (see syllabus for description of course outcomes)

Project Description:

Understand how processes and threads in a Linux system are created and managed by an application. This project is in three parts:

Project Schedule

- Part-0: (Due: **2021-03-01**) Clone, build, execute the iterative version of the project. Study and understand the code. Answer the questions required.
- Part-1: (Due: **2021-03-08**) Complete the process part of the assignment.
- Part-2: (Due: **2021-03-15**) Complete the threads part of the assignment.

This lab pertains to chapter-3 and chapter-4 in your textbook and lecture related to lightweight Linux processes and threads, and the interprocess communication strategies - shared memory models and message passing - used by processes to exchange information.

You are to read in from the standard input two (2) matrices A, of size NxM and B, of size MxP, and then generate a new matrix $A \cdot B$, of size NxP that is the result of multiplying the A and B matrices together.

As a refresh of matrix multiplication:

Let,

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \cdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix},\ B = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \cdots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix}$$

Then,

$$A \cdot B = \begin{pmatrix} (AB)_{11} & (AB)_{12} & \cdots & (AB)_{1p} \\ (AB)_{21} & (AB)_{22} & \cdots & (AB)_{2p} \\ \vdots & \vdots & \cdots & \vdots \\ (AB)_{n1} & (AB)_{n2} & \cdots & (AB)_{np} \end{pmatrix},\ \text{where } (AB)_{ij} = \sum_{k=1}^{m} A_{ik} \cdot B_{kj}$$

Your program must then create a structure for A, for B, and for $A \cdot B$, being stored in a shared memory object to which the parent's children have shared access.

The parent will then create a *row* and *col* set of integer variables and iteratively create *row*-children using the system **fork** call, updating the *row* and *col* integers as needed between each subsequent call to **fork**.

In each child process, the child will use the *row* and *col* values to then compute the,

$$(AB)_{ij} = \sum_{k=1}^{m} A_{ik} \cdot B_{kj}$$

expressions from the A and B shared memory objects, and store the result into the $A \cdot B$ shared memory object in the correct location. Once completed, the child will exit using the system call **exit**. Each child process (or thread) should compute a single row of the $A \cdot B$ matrix.

Once the parent has created all of its children, it will use the system call **wait**, until all of its child processes have terminated, and then it will output the contents of matrix A, the contents of matrix B , and the contents of matrix $A \cdot B$, and exit back to the command shell.

You will also provide a thread-based version using the POSIX *pthreads* library.

Lab Environment

The lab environment will take place on the virtual machine (VM) `csdept11.cs.mtech.edu` . You may ssh into this machine using your normal Department Linux login and password.

Upon logging into the VM, you will be presented your unique `port number` for connection to a VNC session when using the `qemu` emulation software. Remember this port. If you forget the port number, you may type the command `myport` to be presented your port number again. This is the port you will need to use to access emulated OS activities, both internally - from the Campus networks - and externally - from your home or off-campus networks.

Specific Project Tasks:

The following tasks should be completed to obtain the maximum points available for this assignment.

1. Obtaining your project files
    i. Perform the following tasks on the Department GitLab Server:
        a. Login to the Department GitLab server gitlab.cs.mtech.edu with your email address and your password.
        b. Select the project operating-systems/s21-csci460/procsthreads> from the offering of this course.
        c. Click on the **fork** button to create a copy of this project under your own GitLab account.
        d. Go to the **project settings** menu and then the **members** option and add your instructor and any teaching assistants as developers to your fork of the project.
        e. Lastly, go back to this project under your account and click the icon to the right of the project URL. This will copy the project URL to the clipboard so you may paste it in the next sequence of steps.
    ii. Perform the following tasks on the Department Linux server `csdept11.cs.mtech.edu`.
        a. Login to the Department's Linux server with your credentials.
        b. Execute the command(s)

```
cd ~/CSCI460/Projects
```

        c. Execute the command `git clone <url>`, where `<url>` is pasted from the clipboard.
        d. Change into the directory created by the `git clone` command to perform the rest of the tasks for this project.

- Part-0: Explore the provided code and *Makefile* that is used to build three different *targets*: **main-iterative**, **main-process**, and **main-thread**. Each of these targets uses most of the code provided, and only varies based on how the *dot product* is performed, and uses compiler directives to conditionally compile different parts of the code.

    i. Execute the following command(s):

    > cd ~/CSCI460/Projects/procsthreads
    > make

and then, execute the iterative version in one of the two ways below,

> ./main-iterative < test.in
> ./genData -m 3 -e 10 | ./main-iterative

The `genData` program will generate properly formatted input data in the form,

$$\begin{array}{c} n\ m \\ A_{1,1}\ A_{1,2}\ \ldots\ A_{1,m} \\ A_{2,1}\ A_{2,2}\ \ldots\ A_{2,m} \\ \vdots \\ A_{n,1}\ A_{n,2}\ \ldots\ A_{n,m} \\ B_{1,1}\ B_{1,2}\ \ldots\ B_{1,n} \\ B_{2,2}\ B_{2,2}\ \ldots\ B_{2,n} \\ \vdots \\ B_{m,1}\ A_{m,2}\ \ldots\ B_{m,n} \end{array}$$

Where n is the number of rows in matrix A and the number of columns in matrix B, and m is the number of columns in matrix A and the number of rows in matrix B. The content of $A_{i,j}$ cells in matrix A follow, along with the content $B_{i,j}$ cells in matrix B.

ii. Answer the following questions about the *iterative* solution to the problem and place in your lab report file `report.pdf` :

a. what is the purpose of the `ChildData` struct found in main.h?

b. What is the purpose of using `void *` within the code to refer to larger blocks of allocated memory?

c. How are the matrices - `MatrixA` , `MatrixB` , and `MatrixC` - stored in memory?

d. Does the `cptr` variable need to be advanced in the `dotProduct` function, and if so, how does it advance?

e. Would it be appropriate to advance the `cptr` by modifying the code that currently reads,

> *cptr += (*aptr) * (*bptr);

to code that reads,

> *(cptr++) += (*aptr) * (*bptr);

and why, or why not?

- Part-1: Develop the code for the processes part of the project.

   i. Review the code within the `PROCESS` preprocessor directives in the file `main.c`, as well as the code not inside any preprocessor directives.

   ii. Review the code in the file `process/parent.c` which should not need to be modified, but you are free to do so depending on your solution to the problem.

      The code in this file will iterate over the rows of `MatrixC`, setting the `childRow` and `childCol` in the `ChildData` struct as needed. It will then create a child process for each row and keep track of the process-id for the child it creates.

      a. In the child process, the `dotProduct()` function will be called - found in the `process/child.c`. Upon its reture, the `exit()` system call will be invoked, terminating the child process.
      b. In the parent process, a loop will execute once for each child created. Inside the loop the `wait()` system call will be invoked which will not return until a child process terminates. Once all of the child processes have terminated, the `matrixProduct()` will return and the code will begin executing the `main.c`.

   iii. Review the comments in the `process/child.c` for some insights into what needs to be completed.

   iv. Finish implementing the shared memory management - including creating the shared memory object, mapping the shared memory object into the child process' virtual memory, and then performing the matrix calculations and storing the result into the correct locations within the shared memory object.

- Part-2: Develop the code for the thread part of the project.

   i. Similar to Part-1 of the project, but now implemented with *threads* instead of processes, by making use of the *pthreads* library.

   ii. Revisit the code in the `main.c` file and pay attention to the code falling within the `THREAD` preprocessor directives.

   iii. Create and modify any specific data structures that are need for use in this part of the project.

   iv. Implement what is required in `thread/parent.c` file to create child threads to process each row in `MatrixC`.

   v. Implement what is required in `thread/child.c` file to access the required memory where the matrices are stored and compute the specified row for `MatrixC`.

vi. Make sure to do your housekeeping! and keep track of threads created, to join thread back to the main execution thread.

Project Grading:

The project must compile without error (ideally without warnings) and should not fault upon execution. All exceptions should be caught if thrown and handled in a rational manner. Grading will follow the *project grading rubric* shown below:

Attribute (Pts)	**Exceptional (1)**	**Acceptable (0.8)**	**Amateur (0.7)**	**Unsatisfactory (0.6)**
Specification (10)	The program works and meets all of the specifications.	The program works and produces correct results and displays them correctly. It also meets most of the other specifications.	The program produces correct results, but does not display them correctly.	The program produces incorrect results.
Readability (10)	The code is exceptionally well organized and very easy to follow.	The code is fairly easy to read.	The code is readable only by someone who knows what it is supposed to be doing.	The code is poorly organized and very difficult to read.
Reusability (10)	The code only could be reused as a whole or each routine could be reused.	Most of the code could be reused in other programs.	Some parts of the code could be reused in other programs.	The code is not organized for reusability.

Attribute (Pts)	Exceptional (1)	Acceptable (0.8)	Amateur (0.7)	Unsatisfactory (0.6)
Documentation (10)	The documentation is well written and clearly explains what the code is accomplishing and how.	The documentation consists of embedded comments and some simple header documentation that is somewhat useful in understanding the code.	The documentation is simply comments embedded in the code with some simple header comments separating routines.	The documentation is simply comments embedded in the code and does not help the reader understand the code.
Efficiency (5)	The code is extremely efficient without sacrificing readability and understanding.	The code is fairly efficient without sacrificing readability and understanding.	The code is brute force and unnecessarily long.	The cod eis huge and appears to be patched together.
Delivery (total)	The program was delivered on-time.	The program was delivered within a week of the due date	The program was delivered within 2-weeks of the due date.	The code was more than 2-weeks overdue.

Example:

The *delivery* attribute weights will be applied to the total score from the other attributes. If a project scores 36 points total for the sum of *specification*, *readability*, *reusability*, *documentation*, and *efficiency* attributes, but was turned in within 2-weeks after the due date, the project score would be ‘$36 \cdot 0.7 = 25.2$‘.

Collaboration Opportunities:

This project provides *no collaboration opportunities* for the students. Students are expected to design and implement an original solution specific in this project description. Any work used that is not original should be cited and properly references in both the code and in any accompanying write-up. Failure to cite code that is not original may lead to claims of academic dishonesty against the student - if in doubt of when to cite, see your instructor for clarification.