Operating Systems (Code: COMP2006)

CURTIN UNIVERSITY
Department of Computing

Parallel Matrix Multiplication Sum (PMMS)

Due Date: **4PM, Monday May 9, 2016**

Objective

The objective of this programming assignment is to give you some experiences in using multiple processes/threads and their inter-process/thread communications respectively. You will learn how to create processes/threads, shared memory and solve the critical section problems.

Assignment Description

Consider a matrix A of size $M \times N$ and a matrix B size $N \times K$. Let $A_{i,j}$ and $B_{i,j}$ respectively be the entry of matrix A and B at row i and column j. Assume each entry is an integer value. Let C be the matrix product of A and B, i.e., $C = A \times B$; recall that as the result, matrix C has M rows and K columns. Further, if $C_{i,j}$ is the entry of matrix C at row i and column j, we calculate

$$C_{i,j} = \sum_{r=1}^{N} A_{i,r} \times B_{r,j}$$

In other words, each $C_{i,j}$ is the sum of the products of elements for row i in matrix A and column j in matrix B. For example, if A is a 3×2 matrix, and B is a 2×4 matrix, C would be a 3×4 matrix, and $C_{3,2} = A_{3,1} \times B_{1,2} + A_{3,2} \times B_{2,2}$. More specifically, given

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \text{ and } B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}, \text{ we compute}$$

$$C = A \times B = \begin{bmatrix} 1\times1 + 2\times5 & 1\times2 + 2\times6 & 1\times3 + 2\times7 & 1\times4 + 2\times8 \\ \mathbf{3\times1 + 4\times5} & \mathbf{3\times2 + 4\times6} & \mathbf{3\times3 + 4\times7} & \mathbf{3\times4 + 4\times8} \\ 5\times1 + 6\times5 & 5\times2 + 6\times6 & 5\times3 + 6\times7 & 5\times4 + 6\times8 \end{bmatrix}$$

$$= \begin{bmatrix} 11 & 14 & 17 & 20 \\ \mathbf{23} & \mathbf{30} & \mathbf{37} & \mathbf{44} \\ 35 & 46 & 57 & 68 \end{bmatrix}$$

In this project, you are asked to compute matrix C and compute the sum of entries in matrix C, denoted as ***total***, in parallel using M processes/threads. Specifically, (i) process i computes the K entries in row i of matrix C, and calculates the total value of the entries, denoted as $\boldsymbol{subtotal_i}$ for $i = 1, 2, \ldots, M$; (ii) the parent process/thread computes $\boldsymbol{total} =$

$\sum_{i=1}^{M} \boldsymbol{subtotal_i}$. In the example, process i = 2 computes the K = 4 **bolded** entries in matrix C, and $\boldsymbol{subtotal_2}$ = 23 + 30 + 37 + 44 = 134, and ***total*** = 62 + 134 + 206 = 402.

Implementation

Part A: using processes (50%)

Write a program in C language in which the main process (i) creates M child processes, each of which computes $\boldsymbol{subtotal_i}$, (ii) waits for the subtotal from each of the M child processes, (iii) computes $\boldsymbol{total} = \sum_{i=1}^{M} \boldsymbol{subtotal_i}$, and (iv) print each subtotal as well as **total**. For the example, the program prints the following output (x_i is the process ID of process i):

Subtotal produced by process with ID x_1: 62
Subtotal produced by Processor with ID x_2: 134
Subtotal produced by Processor with ID x_3: 206
Total: 402

Let us call the executable for the parent process as **pmms**. The program should be run as:

pmms *matrix_A matrix_B M N K*

where *matrix_A* and *matrix_B* are the name of files that contain the entries of matrix A and matrix B respectively, and M, N, K specify the dimensions of the two matrices. For the example, the contents of the two matrices are as follows; the integers in each line are separated by a space.

matrix_A	*matrix_B*
1 2	1 2 3 4
3 4	5 6 7 8
5 6	

The details of the implementation are as follow.

1) The parent process (i) reads the integers in files *matrix_A* and *matrix_B*, (ii) puts the entries in two 2-dimensional arrays A and B of size $M \times N$ and $N \times K$ respectively, (iii) creates a two dimensional array C of size $M \times K$, and (iv) creates a data structure named **subtotal** to store the subtotal generated by each child process and its process ID. The **subtotal** can store ONLY ONE (1) subtotal at a time.

2) The parent process (i) creates M child processes, and (ii) reads any available **subtotal** produced by each of the M child processes.

   Since processes do not share memory, you need to make arrays A, B and C each as a shared memory block so that the M processes and the parent process can access them. Similarly, **subtotal** is another shared memory block.

   Note that you can consider accesses to **subtotal** as the producer-consumer problem in which the main process is the consumer and each of the M processes the producer.

3) Each child process i, for i = 1, 2, …, M, (i) computes the K entries in row i of array C from arrays A and B, (ii) calculates $\boldsymbol{subtotal_i}$, and (iii) puts $\boldsymbol{subtotal_i}$ and its ID into **subtotal**.

   Accesses to **subtotal** must be synchronized. Specifically, each process should put its subtotal and ID only when **subtotal** is *empty*; i.e., the content of **subtotal** has been *consumed* by the parent process. Similarly, the parent process should read **subtotal** only when it contains a new subtotal.

4) The parent process (i) prints each subtotal and the ID of process that computed it, (ii) the sum of all subtotals, i.e., ***total***, and (iii) terminates.

   Make sure that before the parent process terminates, it does not leave behind any zombie processes and other system resources, i.e., shared memory and semaphore variables.

Please read Chapter 3 of the textbook (Operating System Concepts by Silberschatz, et al.) to learn how to create shared memory, and Chapter 6 of the same textbook on how to use POSIX semaphores.

Part B: using threads (30%)

Repeat **Part A**, except that you use threads. Note that threads share memory, and thus you need not explicitly create *shared memory* for arrays A, B and C, and **subtotal**. However, you have to address the same synchronization issues.

Please read Chapter 4 of the textbook (Operating System Concepts by Silberschatz, et al.) to learn how to create POSIX threads using pthread_create(), and Chapter 6 of the same textbook on how to use POSIX pthread_mutex_lock(), pthread_mutex_unlock(). You also need to use pthread_cond_wait() and pthread_cond_signal().

Instruction for submission

1. Assignment submission is **compulsory**. NO LATE SUBMISSION WILL BE ACCEPTED. If your assignment is incomplete due to illness or other circumstances on or near the submission date you must submit the partial solution you have completed. Any justified exceptional circumstance (e.g., due to illness supported with a Medical Certificate) will be taken into consideration.

2. You must (i) submit a hard copy of your assignment report to the unit box, (ii) submit the soft copy of the report to the unit Blackboard (in one **zip file**), and (iii) put your program files i.e., pmms.c, makefile, and other files, e.g., test input, in your home directory, under a directory named **OS/assignment**.

3. Your assignment report should include:
   - A signed cover page that includes the words "Operating Systems Assignment", and your name in the form: family, other names. Your name should be as recorded in the student database.
   - Software solution of your assignment that includes (i) all source code for the programs with proper in-line and header documentation. Use proper indentation so that your code can be easily read. Make sure that you use meaningful variable names, and delete all unnecessary comments that you created while debugging your program; and (ii) readme file that, among others, explains how to compile your program and how to run the program.
   - Detailed discussion on how any mutual exclusion is achieved and what processes/threads access the shared resources.
   - Description of any cases for which your program is not working correctly or how you test your program that make you believe it works perfectly.
   - Sample inputs and outputs from your running programs.

   Your report will be assessed (worth 20% of the overall assignment mark).

4. Due dates and other arrangements may only be altered with the consent of the majority of the students enrolled in the unit and with the consent of the lecturer.

5. Demo requirements:
   - You may be required to demonstrate your program and/or sit a quiz during tutorial sessions (to be announced).
   - For demo, you MUST keep the source code of your programs in your home directory, and the source code MUST be that submitted. The programs should run on any machine in the department labs.

Failure to meet these requirements may result in the assignment not being marked.