Lab 3: Thread Synchronization and Thread Parallelization Using Pthreads

IUPUI CSCI 503 – Fall 2013
Assigned: September 23, 2013
Due time: 11:59PM, October 7, 2013

1. Introduction

You have learned in classes spin locks, mutex locks, semaphores, monitors, and condition variables. Now it's time to practice what you have learned to gain hands-on experience. Only understanding the thread APIs and example codes are far from enough. In this lab, you will solve two real problems: (1) the classic multiple producer–multiple consumer problem, and (2) the parallel matrix multiplication problem.

2. Multiple Producer–Multiple Consumer Problem

In the lecture, I talked about how to use semaphores to solve the producer-consumer problem. In fact, the solution supports multiple producer threads and multiple consumer threads. So your first task is to use pthread mutex and pthread condition variable to implement your own version of semaphore that supports P() and V() operations. Next, use your own semaphore to write a parallel program that can create M producer threads and N consumer threads. Each producer sleeps for a random number of seconds (to pretend to "produce" a product), then adds a new item to a shared ring buffer. Similarly, each consumer removes an item from the shared ring buffer, then sleeps for a random number of seconds (to pretend to "consume" a product).

While producing or consuming, each thread should keep track of how many products have been produced or consumed. This way, you can tell if your program is working or not. For instance, if the total number of consumed products is less than the total number of produced products, your program has problems.

Your executable program should take five arguments **exactly in the following order**: Number_Producers, Number_Consumers, Max_Sleep_Seconds, Total_Number_Items2Produce, and Ring_Buffer_Size.

Each producer or consumer can sleep between 1 second and MAX_Sleep_Seconds.

3. Parallel Matrix Multiplication Problem

You must use pthreads to compute C = A*B, where A, B, C are NxN square matrices. N is the matrix size. You need to implement a sequential version first in order to verify your multithreaded version. The sequential version is the simplest one which has three for loops: i, j, k. All the matrix elements are double floating point values (i.e., a type of double).

In the multithreaded version, there are a number of N^2 C[i, j] elements that need to computed. Your program should create m threads. Each thread will pick up one of the N^2 C[I,j] elements to compute at runtime. You must make sure that each element C[i,j] is computed exactly once by one thread.

Your executable command should take two arguments in the following given order: Matrix_Size and Number_Threads. Also, remember to compare the result of the multithreaded version with the result of the sequential version. If their results are the same, print SUCCESS. Otherwise, print an error message.

4. Score Distribution

- Coding style: **10%** (as described in the "Labs Grading Policy")
- Multiple Producer – Multiple Consumer: **50%** (correct semaphore implementation: 20%, correct MPMC implementation: 30%).
    - Your MPMC will be tested using different number of producers/consumers, and different buffer sizes.
- Parallel Matrix Multiplication: **40%** (correct sequential version: 10%, correct parallel implementation: 30%).
    - Different matrix size and different number of working threads will be tested.

Again, if there is a "Segmentation Fault" when testing your code, then TA will deduct 20 points from your score.

The total score is 100 points.

5. Deliverable

There should be two subdirectories: one for each version.

In each subdirectory, there should be source code in C, a Makefile, and a README. Make a tar ball for the above two subdirectories and send it to the TA. The tar ball name may be "Lab3_your_name.tar".