#### COEN 177: Operating Systems

**Lab assignment 4: Developing multi-threaded applications**

**Objectives**

##### To develop multi-threaded application programs

##### To demonstrate the synchronized and unsynchronized use of threads (in simple matrix manipulations)

##### **Guidelines**

##### This assignment is adapted from examples in one of the class text books,[[1]](#footnote-0) which noted how the increasing importance of parallel processing led to the development of lightweight user-level thread implementations. Even so, the topic of whether threads are a better programming model than processes or other alternatives remains open. Several prominent operating systems researchers have argued that normal programmers should almost never use threads because (a) it is just too hard to write multi-threaded programs that are correct and (b) most things that threads are commonly used for can be accomplished in other, safer ways. These are important arguments to understand—even if you agree or disagree with them, researchers point out pitfalls with using threads that are important to avoid. The most important pitfall is the concurrent access of threads to shared memory. When threads concurrently read/write to shared memory, program behavior is undefined. This is because a thread schedule on the CPU is non-deterministic. The program behavior completely changes when you rerun the program. This should have been obvious to you by now with the examples you have run in prior labs.

##### To ensure a deterministic behavior of programs where threads cooperate to access shared memory, a synchronization mechanism needs to be implemented. Consequently,

##### The program behavior will be related to a specific function of input, not of the sequence of which thread runs first on the CPU

##### The program behavior will be deterministic and will not vary from run to run

##### These facts should not be IGNORED, otherwise the compiler will mess up the results and will not be according what is thought would happen

##### This lab is designed to give you the first hands-on programming experience on developing multi-threaded applications. In the coming labs, you will

##### 

**C Program with threads (drawn from problems 1, 2, 7, and 8 of the aforementioned class textbook)**

In chapter 4 of the class text book, the threadHello.c program has been discussed. The threads run on the CPU in different orders. Demonstrate each of the following steps to the TA to get a grade on this part of the lab assignment.

1. Download the slightly revised threadHello.c program from Camino, then compile and run several times. The comment at the top of the program explains how to compile and run the program.

Explain what happens when you run the threadHello.c program? Do you get the same result if you run it multiple times? What if you are also running some other demanding processes (e.g., compiling a big program, playing a Flash game on a website, or watching streaming video) when you run this program?

The function go() has the parameter arg passed a local variable. Are these variables per-thread or

shared state? Where does the compiler store these variables’ states?

The main() has local variable i. Is this variable per-thread or shared state? Where does the compiler store this variable?

1. Delete the second for loop in threadHello.c program so that the main routine simply creates NTHREADS threads and then prints “Main thread done.” What are the possible outputs of the program now. Explain.

**Matrix transposition with threads (adapted from problem 5 of the same textbook)**

1. Write a program that uses threads to copy, and update, a matrix in parallel. The goal is to copy the contents of one matrix to another, and to update the original matrix (by multiplying each element by 2).

You may fill in the entries of A and B matrices (double matrixA[N][M], matrixB[M][L] ) using a predefined sequence as below:

x = 0;

for (int i = 0; i < N; i++)

for (int j = 0; j < M; j++)

matrixA[i][j] = x++;

for (int i = 0; i < N; i++)

for (int j = 0; j < M; j++)

matrixB[i][j] = 0;

This creates two matrices, A and B, with A containing a simple sequence of numbers, and B initialized to zeros.

The following are important notes:

* The values of N and M must be large to exploit parallelism (e.g. N, M = 1024).
* Implement separate functions to print out a matrix, to copy matrixA to matrixB, and to double the values of each element in matrix A.
* Do the following:
  + print matrixA.
  + launch two separate threads to:
    - copy matrixA to matrixB
    - double the values in matrixA
  + wait for the two threads to complete
  + print matrixA again.
* The main thread needs to wait for the two subroutine threads to complete, but in your first implementation there should be no synchronization between the threads (i.e., do nothing to coordinate their behavior relative to each other).
* Repeat the above, but ensure that the copying of the matrix does not start until the doubling is complete (**hint:** you may use a spin-lock/loop if you wish, but clearly explain how you are ensuring that the two independent threads wait appropriately).

**Requirements to complete the lab**

1. Show the TA correct execution of the C programs.
2. Submit your answers to questions, observations, and notes as .txt file and upload to Camino
3. Submit the source code for all your programs as .c file(s) and upload to Camino.

Be sure to retain copies of your .c and .txt files. You will want these for study purposes and to resolve any grading questions (should they arise)

Please start each program/ text with a descriptive block that includes minimally the following information:

# Name: <your name>

# Date: <date> (the day you have lab)

# Title: Lab4 – task

# Description: This program computes … <you should

# complete an appropriate description here.>

1. J. Anderson and M. Dahlin, Operating Systems – Principles and Practice, Recursive Books, 2nd Edition, 2014 [↑](#footnote-ref-0)