**CSC 338 Parallel and Distributed Computing**

**Exercise No. 8, April 17, 2017**

**Pthreads and Critical Sections**

**Goal**

Learn how to handle critical sections with pthreads

**Background**

A critical section is a section of code that can’t be executed by multiple threads simultaneously.

**Procedure**

1. Compile *pth\_pi.c,* located in the exercise folder on the class server:

gcc -g -Wall -o pth\_pi pth\_pi.c -lm –lpthread. The program implements another way to estimate pi—by adding an infinite series of fractions. The program is essentially a global sum, in which each thread is responsible for summing a part of the series and the total sum is a shared (global) variable. The number of threads and number of terms to use are command line arguments. Execute the program with several numbers of threads and terms. Invoke the program as follows:

./pth\_pi number\_of\_threads number\_of\_terms.

The program estimates pi using threads and also computes a serial estimate, displaying both. We expect the two estimates to be the same but you should notice they are not because this program has a serious bug. Adding numbers in a high level language is not an atomic operation—the total and the number added to it must be loaded into registers, added, then the result stored back to a memory location. So the statement adding terms to *sum* is a critical section—it should only be executed by one thread at a time.

1. One simple solution to this problem is to keep each thread busy-waiting until its turn to access the shared variable. Add code to pth\_pi.c so each thread can access *sum* safely: before carrying out the sum use a while loop with an empty body that waits until a shared variable *(flag)* is equal to the rank of the thread. When the thread has carried out the addition, it should increment *flag.* You may realize that you don’t want *flag* to take a value higher than *thread\_count – 1.* The simplest way to do that is using modulo:

*flag = (flag + 1) % thread\_count;*

Compile the modified program and execute it several times with various numbers of threads and terms. Are the serial and threaded estimates consistent?

1. Consider that busy-waiting inside the loop might waste a significant amount of time. If you have a private (local) variable that a thread can use inside the loop to calculate its own local sum, then add that to the shared global sum after the loop, then you only have to protect the addition to *sum* after the loop. Make those changes then compile and run the program several times to see if you still get good estimates.