Shared Memory Programming with Pthreads
Programming Assignment – 4

Exercise – 1 [2 points] – Monte Carlo [Pthread implementation]

Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are 0.6m in length. Suppose also that there's a circle inscribed in the square dartboard. The radius of the circle is 0.3m, and it's area is 0.093π square meters. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation:

$$\text{number in circle / total number of tosses} = \pi/4$$

since the ratio of the area of the circle of the square is π/4. We can use this formula to estimate the value π with a random number generator:

```
number in circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
x = random double between − 1 and 1;
y = random double between − 1 and 1;
distance_squared = x ∗ x + y ∗ y;
if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4 ∗ number_in_circle/((double) number_of_tosses);
```

Write a Pthreads program that uses a Monte Carlo method to estimate π. The main thread should read in the total number of tosses and print the estimate. You may want to use **long long int** for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of π.

Exercise – 2 [2 points]

During the lecture sessions we examined how to perform the π calculation in a shared memory system using mutexes. Run and examine the performance of this program when we increase the number of threads beyond the number of available CPUs. What do you observe? What does this suggest about how the threads are scheduled on the available processors? Your answer should be based on measurable results.

Exercise – 3 [6 points]

During the lecture sessions we examined how to perform the π calculation in a shared memory system using mutexes:

a) Modify the mutex version of the π calculation program so that the critical section is in the `for` loop. How does the performance of this version compare to the performance of the original busy-waiting version? How might we explain this?

b) Modify the mutex version of the π calculation program so that it uses semaphore instead of a mutex. How does the performance of this version compare with the mutex version?

Show measurable results to support the answer that you give to the above questions.