CS 302 – Assignment #6

Purpose: Learn concepts regarding threading and data races
Due: Tuesday (10/17) → Must be submitted on-line before class.
Points: Part A → 50 pts, Part B → 25 pts

Assignment:

Part A:

In recreational number theory, a Happy Number[1] is a number defined by the following process: starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number either equals 1 (where it will stay), or it loops endlessly in a cycle that does not include 1. Those numbers for which this process ends in 1 are referred to as happy numbers, 😀 while those that do not end in 1 are unhappy ☹ numbers (or sad numbers).

JOE, HAVE YOU READ THIS SECTION?
YEAH. I KNOW IT ALL.
MATH
5-3
OKAY. WHAT ARE THE WHOLE NUMBERS?
MATH
UH, 8, 6, 9, AND 0!
©2014 Rick Detorie. dist. by Creators.com
REALLY?
YEAH. THEY'RE THE ONES WITH HOLES IN 'EM!
MATH
detorie

For example, 19 is happy, as the associated sequence is:

$$\mathbf{1^2 + 9^2 = 82}$$
$$\mathbf{8^2 + 2^2 = 68}$$
$$\mathbf{6^2 + 8^2 = 100}$$
$$\mathbf{1^2 + 0^2 + 0^2 = 1}$$

It turns out that all unhappy numbers have a 4 in the endless sequence. This allows us to stop when the process results in a 4.

Write a threaded C++11 program to provide find the count of happy and sad numbers between 1 and a user provided limit (inclusive). For example, between 1 and 1,000 there are exactly 143 happy numbers and 857 sad numbers.

In order to improve performance, the program should use threads to perform computations in parallel. The program should read the thread count and limit from the command line in the following format:

```
./happyNums -t <threadCount> -l <limitValue>
```

Additionally, the program should use the C++11 high resolution clock to provide an execution time in milliseconds. *Note*, you will need to create your own source file and makefile.

Part B:

When completed, use the provided timing script, **`ast6Exec`**, to execute the program various times with single and multiple threads (>30 minutes). The script writes the results to a file (a6times.txt). To ensure consistent results, use the **`cssmp.cs.unlv.edu`** which has 32 cores (uses CS login). Enter the thread counts and times into a spreadsheet and create a line chart plot of the execution times versus the thread counts. Refer to the example below for how the plot should look.

[1] For more information, refer to: https://en.wikipedia.org/wiki/Happy_number

Note, the default g++ compiler version on CSSMP does not support the C++11 standards. In order to use the current C++11 standard, you will need to type the following (once per session):

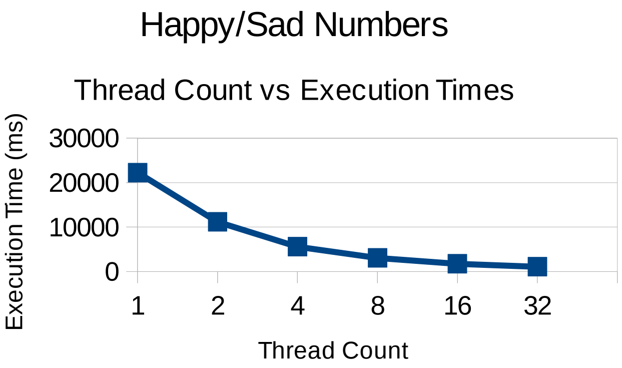
```
scl enable devtoolset-2 bash
```

Create and submit a brief write-up (PDF format), not to exceed ~500 words, including the following:

- Name, Assignment, Section.
- Summarize the results from the timing script including a copy of the timing script output and a copy of the chart.
- Determine the percentage speed-up[2] using the below formula (for each thread count > 2):

$$speedUp = \frac{Time_{single}}{Time_{new}}$$

- Remove the mutex locks and execute the program. Report the results. Include a description of what happened and an explanation of why.



Next Numbers to Check

In order to ensure efficiency and maximum thread utilization, the next set of numbers should be obtained from a function. Each thread function should check 100 numbers at a time. When done, the thread function should request the next set of 100 numbers until all numbers up to (and including) the limit have been checked. You must use a mutex on the global variable for tracking the next set of numbers.

Submission:

When complete, submit:

- Part A → A copy of the **source files** via the class web page (assignment submission link) by class time on the due date. The source files, with an appropriate *makefile*, should be placed in a ZIP folder. The grader will download, uncompress, and type **make** (so you must have a valid, working *makefile*).
- Part B → A copy of the write-up including the chart (see example). Must use PDF format.

Assignments received after the due date/time will not be accepted. Make sure your program includes the appropriate documentation. See Program Evaluation Criteria for CS 302 for additional information. ***Note, points will be deducted for especially poor style or inefficient coding.***

[2] For more information, refer to: https://en.wikipedia.org/wiki/Speedup

Threads

C++11 introduced a new thread library. This library includes utilities for starting and managing threads. It also contains utilities for synchronization like mutexes and other locks, atomic variables and other concurrency related functions.

When you create an instance of a **std::thread**, it will automatically be started. When you create a thread you have to give it the code by passing the function as a pointer. For example, the following is a very common (but not useful) threaded Hello World program:

```
#include <iostream>
#include <thread>

using namespace std;

void helloFunc()
{
    cout << "Hello from thread" << endl;
}

int main(){
    thread thd1(helloFunc);              // start thread
    thd1.join();                         // wait for thread to finish

    return 0;
}
```

As shown, the ***join()*** is used to wait for the thread to complete. You must wait for all threads to complete before terminating the program. You can dynamically create an array of threads in order to effectively handle a varying number of threads.

In addition, mutex's must be used to avoid race data races. To use a mutex, first declare a mutex variable (global in this example) and use the mutex variable as shown in order to avoid race conditions.

```
mutex   myMutex;

void increment(){
    lock_guard<mutex> guard(myMutex);
    // perform applicable increment
}

void decrement(){
    lock_guard<mutex> guard(myMutex);
    // perform applicable decrement;
}
```

The **lock_guard<mutex> guard(mutexVariable)** operation will ensure that the subsequent statements are locked and thus only executed by one thread at a time. Specifically, if the mutex is free, it will be locked and the the subsequent code will executed. If the mutex is already locked, all other threads will be blocked from executing the code until one obtains the lock. The lock will be released when it goes out of scope (at the end of the function).

To compile the, include the **-std=c++11** option to get the C++11 support activated. Additionally include the **-pthread** option for the threading libraries. You will need to **#include <thread>** and **#include <mutex>** as part of the program includes.

C++11 High Resolution Timer

The functions for the C++11 high resolution timer are located in the chrono library.

```
// get start timer...
auto t1 = chrono::high_resolution_clock::now();

// do stuff...

// get end time...
auto t2 = chrono::high_resolution_clock::now();

// show results with times
cout << "Program took: " <<
      << std::chrono::duration_cast<std::chrono::milliseconds>
            (t2 – t1).count() << " milliseconds" << endl;
```

You will need to **`#include <chrono>`** as part of the program includes.