CS 370 – Threading Project

Purpose: Become more familiar with operating system interaction, threading, and race conditions.
Points: 120 (60 for program and 60 for write-up)
Scoring will include functionality, documentation, and coding style

Assignment:

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).

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

$$1^2 + 9^2 = 82$$
$$8^2 + 2^2 = 68$$
$$6^2 + 8^2 = 100$$
$$1^2 + 0^2 + 0^2 = 1$$

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 C language program provide a count of the happy and sad numbers between 1 and some user provided limit. In order to improve performance, the program should use threads to perform some of the computations in parallel. The program should read the thread count option and limit the command line.

For example, between 2 and 100,000 there are exactly X happy numbers and Y sad numbers. In order to improve performance, the program should use threads to perform computations in parallel.

A template is provided to help parse the command line arguments (described on the next page).

When the program is complete, create a bash script file that executes the program a series of times using each using one (1), two (2), three (3) and four (4) threads. The Unix time command should be used to obtain the execution times. We will use the "real" time. The script should execute and time each with a limit of 10,000,000. The final step will be to chart and explain the results

The reference, POSIX thread (pthread) libraries, may be useful along with the C Thread Example (last page).

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

Command Line Arguments:

The program should read the number of threads and and use limit from the command line in the following format; "./happyNums -t <1|2|3|4> -l <limitValue>". For example:

```
      ./happyNums -t <1|2|3\4> -l 10000000
```

If either option is not correct, an applicable error message should be displayed. The minimum acceptable limit is 100. If no arguments are entered, a usage message should be displayed. Refer to the sample executions for examples of the appropriate error messages.

Locking

When changing global variables, a mutex must be used. For this usage, the mutex variables are declared globally.

```
        // global declaration
        pthread_mutex_t      myLock;    // mutex variable


        // initialize myLock mutex.
        pthread_mutex_init(&myLock, NULL);


        // code section
        pthread_mutex_lock(&myLock);
        // critical section code
        pthread_mutex_unlock(&myLock);
```

The program should use two mutexes; one mutex for the global counter and one for the counters.

Script File

Create a bash script file that executes the program a series of times using each using one (1), two (2), three (3) and four (4) threads. The program executable should be passed an argument. The Unix time command should be used to obtain the execution times. We will use the "real" time. The script should execute and time each with a limit of 10,000,000. You should execute the script a series of times and ensure that the results are consistent. When done, choose one representative set for use in the results write-up.

Results

When the program is working, complete additional timing and testing actions as follows;

- Use the bash script you created to execute and time the working program.
  - *Note*, you may halve the 10,000,000 if the execution times are too long.
- Compute the speed-up[2] factor from the base sequential execution and the parallel execution times. Use the following formula to calculate the speed-up factor:

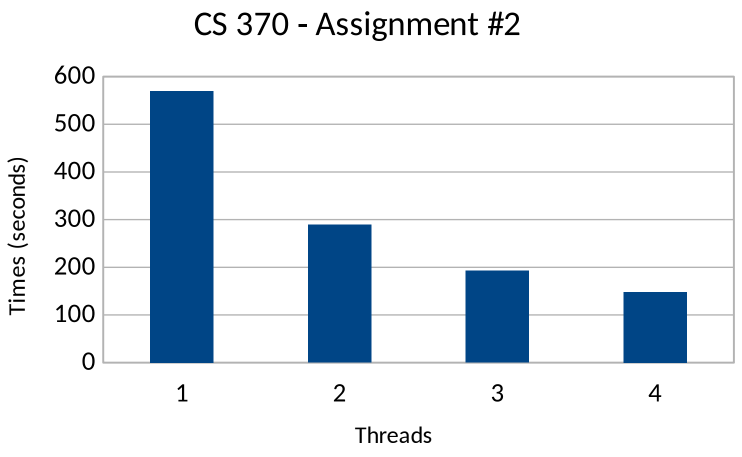
$$\mathbf{\textit{SpeedUp}} = \frac{\mathbf{\textit{ExecTime}}_{\textit{sequential}}}{\mathbf{\textit{ExecTime}}_{\textit{parallel}}}$$

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

- Remove the locking calls (the 'pthread_mutex_lock' calls and the 'pthread_mutex_unlock') and re-execute the program using a limit of 10,000,000 for 1 thread and then 4 threads.

- Create a final write-up including a copy of the program output from the timing script and an explanation of the results. The explanation must address
  - the final results for 1, 2, 3, and 4 thread executions, including final results (list of amicable numbers) and the execution times for both each.
  - the speed-up factor from 1 thread to 2, 3, and 4 threads (via the provided formula)
  - simple chart plotting the execution time (in seconds) vs thread count (see example below)
  - the difference with and without the locking calls for the parallel execution
    - explain specifically what caused the difference (if any)

The explanation part of the write-up (not including the output and timing data) should be less than ~300 words. Overly long explanations will not be scored.



Submission:
When complete, submit:

- A copy of the **C** source code (not C++) file.

- Submit a write-up (PDF format) including the following topics:
  - Copy of the program output from the script file (above)
    - An explanation of the results (sequential and parallel) with calculated speed-up.
  - When the program is working, comment the lock and unlock calls and execute.
    - Report the results.
    - The different with and without the locking calls for the parallel execution
  - *Note*, overly long explanations will not be scored.

- A copy of the bash script use for the timing.

Note, the program must compile and execute under Ubuntu 20.04 LTS (and will only be testing with Ubuntu). Assignments not executing under Ubuntu will not be scored. See additional guidance on following pages.

Submissions received after the due date/time will not be accepted.

Example Execution:
The following are some example executions, including the required error checking:

```
ed-vm%
ed-vm% time ./happyNums -t 1 -l 10000000
Count of Happy and Sad numbers from 1 to 10000000
Please wait. Running...

Count of Happy Numbers = ??
Count of Sad Numbers = ??
real  2m11.053s
user  2m11.012s
sys   0m0.036s
ed-vm%
ed-vm%
ed-vm% time ./happyNums -t 3 -l 10000000
Count of Happy and Sad numbers from 1 to 10000000
Please wait. Running...

Count of Happy Numbers = ??
Count of Sad Numbers = ??

real  0m33.668s
user  2m14.147s
sys   0m0.360s
ed-vm%
ed-vm%
ed-vm% ./happyNums -t 3
Error, invalid command line arguments.
ed-vm%
ed-vm%
ed-vm% ./happyNums -t 3 -l 1
Error, limit must be > 100.
ed-vm%
ed-vm%
ed-vm% ./happyNums -t 3 -l -100
Error, invalid limit value.
ed-vm%
ed-vm%
ed-vm% ./happyNums -t five -l 10000000
Error, invalid thread count.
ed-vm%
```

Note, the timing shown is for one specific machine. Actual mileage may vary.

C Thread Example

```c
// CS 370 C Thread Example. Useless example of how threads are used in C.
// use:  gcc -Wall -pedantic -g -pthread -o threadExp threadExp.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

//  Global variables
pthread_mutex_t             myLock;       // mutex variable
int                         j = 0;        // global shared variable

//  Thread function, just prints the global variable five times.
void * do_process() {
   int i = 0;

   pthread_mutex_lock(&myLock);
   printf("\nThread Start\n");

   j++;

   while(i < 5) {
            printf("%d", j);
            sleep(0.5);
            i++;
   }

   printf("Thread Done\n");
   pthread_mutex_unlock(&myLock);

   return    NULL;
}

int main(void)
{
   unsigned long int       thdErr1, thdErr2, mtxErr;
   pthread_t        thd1, thd2;

   printf("C Threading Example.\n");

   //  Initialize myLock mutex.
   mtxErr = pthread_mutex_init(&myLock, NULL);

   if (mtxErr != 0)
            perror("Mutex initialization failed.\n");

   //  Create two threads.
   thdErr1 = pthread_create(&thd1, NULL, &do_process, NULL);
   if (thdErr1 != 0)
            perror("Thread 1 fail to create.\n");

   thdErr2 = pthread_create(&thd2, NULL, &do_process, NULL);
   if (thdErr2 != 0)
            perror("Thread 2 fail to create.\n");

   //  Wait for threads to complete.
   pthread_join(thd1, NULL);
   pthread_join(thd2, NULL);

   //  Threads done, show final result.
   printf("\nFinal value of global variable: %d \n", j);

   return    0;
}
```