**CS 218 – Assignment #12**

Purpose: Become more familiar with operating system interaction, threading, and race conditions.
Points: 100 10 – Header and comments, 45 – Program, and 45 – Write-up
Scoring will include functionality, documentation, coding style, and write-up

**Assignment:**

In recreational number theory, a Happy Numbers[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 an assembly language program to find the count of 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 number limit in base 13 from the command line in the following format; "./happyNums -t<1|2|3|4> -lm <base13Number>". For example:

```
./happyNums -t4 -lm 41B3427
```

The provided C++ main program calls the following routines:

- A value returning function, **getCommandLineArgs()**, that reads and verifies the command line arguments. This includes error and range checking based on provided parameters (in the template). The error strings are predefined in the template. The function should call the **base13toint()** function (you can change the name of this function).
- A void function, **findHappyNumbers()**, that will be called as a thread function and determine of a number of happy or sad and update a global counter (for each). The thread must perform updates to the global variables in a critical section to avoid race conditions. See below sections for additional detail.

A main program and a functions template will be provided. All functions must be in a separate source file, a11procs.asm, that is independently assembled. Only the functions file, not the provided main, will be submitted on-line. As such, you must not change the provided main! You may use static variables as needed (no requirement to create stack-dynamic locals).

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

**Submission:**

- All source files must assemble and execute on Ubuntu and assemble with **`yasm`**.
- Submission three (3) files
  - Submit Source File
    - *Note*, only the functions file (**`a12procs.asm`**) will be submitted.
  - Submit Timing Script Output
    - Submit the **`a12times.txt`** file (created after executing the **`a12timer`** script).
  - Submit Write Up file
    - Includes system description, percentage change, and results explanation (per above).
- Once you submit, the system will score the code part of the project.
  - If the code does not work, you can (and should) correct and resubmit.
  - You can re-submit an unlimited number of times before the due date/time.
- Late submissions will be accepted for a period of 24 hours after the due date/time for any given lab. Late submissions will be subject to a ~2% reduction in points per an hour late. If you submit 1 minute - 1 hour late -2%, 1-2 hours late -4%, … , 23-24 hours late -50%. This means after 24 hours late submissions will receive an automatic 0.

**Program Header Block**

All source files must include your name, section number, assignment, NSHE number, and program description. The required format is as follows:

```
;  Name: <your name>
;  NSHE ID: <your id>
;  Section: <section>
;  Assignment: <assignment number>
;  Description: <short description of program goes here>
```

Failure to include your name in this format will result in a loss of up to 3%.

**Scoring Rubric**

Scoring will include functionality, code quality, and documentation. Below is a summary of the scoring rubric for this assignment.

| Criteria | Weight | Summary |
|---|---|---|
| Assemble | - | Failure to assemble will result in a score of 0. |
| Program Header | 3% | Must include header block in the required format (see above). |
| General Comments | 7% | Must include an appropriate level of program documentation. |
| Program Functionality (and on-time) | 45% | Program must meet the functional requirements as outlined in the assignment. Must be submitted on time for full score. |
| Write-Up | 45% | Write-up includes required section, percentage change is appropriate for the machine description, and the explanation is complete. |

**Thread Function:**
Create a thread function, ***findHappyNumbers()***, that performs the following operations:

- Obtain the next group, COUNT_SET size, of numbers to check (via global variable, ***idxCounter***).
- While the next number group is ≤ ***numberLimit*** (globally available);
  - Check for happy/sad.
  - If happy, increment the *happyCount* variable, if sad, increment the *sadCount* variable.

When obtaining the next group of numbers and updating (+COUNT_SET) the ***idxCounter*** variable, must be performed as a critical section (i.e., locked and unlocked using the provided *spinLock()* and *spinUnlock()* functions). As the numbers are being checked, no locks are needed. Once a number has been determined to be happy or sad, the appropriate counter should locked (via lock prefix) and incremented accordingly. For example:

```
lock    inc    qword [happyCount]
```

It is recommended to complete and test the program initially with only the single sequential thread before testing the parallel threads. In order to debug, you may wish to temporarily insert a direct call (non-threaded) to the ***findHappyNumbers()*** function.

**Results Write-Up**
When the program is working, complete additional timing and testing actions as follows;

- Use the provided script file to execute and time the working program.

- 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:

$$SpeedUp = \frac{ExecTime_{sequential}}{ExecTime_{parallel}}$$

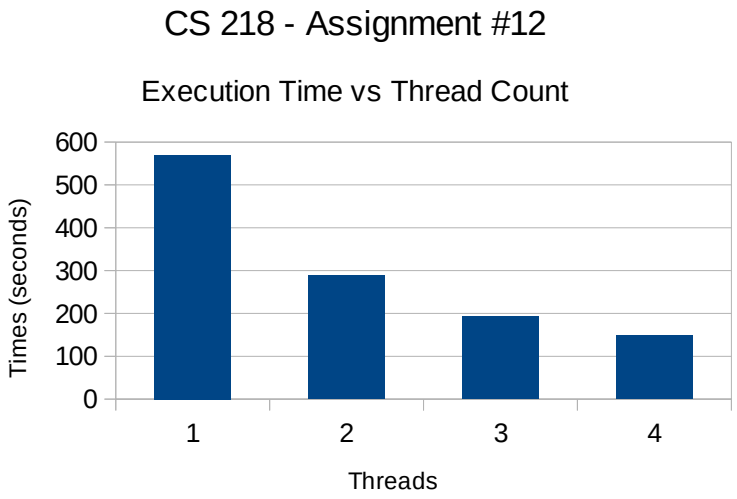
- Remove the locking calls (the spinLock calls and the 'lock') and re-execute the program using a limit of 40,000,000 ($8396851_{13}$) for 1 thread and then 4 threads.
  - The Unix time command (as per asst #11B) should be used to obtain the execution times. Use the "real" time.

- Create a final write-up including a copy of the program output from the timing script file 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

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

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

Below is an example of the type of chart to include




Such graphs are most easily done using a spreadsheet. An optional example spreadsheet is provided for reference.

**Assignment #12 Timing Script**

In order to obtain the times for the write-up a timing script is provided. After you download the script file, **`a12timer`**, set the execute permission as follows:

```
ed-vm$ chmod +x a12timer
ed-vm$ ./a12timer happyNums
```

The script file will expect the name of the executable as an argument (as shown).

The script may take a while to execute, but no interaction is required. The script will create an output file, **`a12times.txt`**, which will be included in the submission and the data be used create the write-up.

**Example Execution:**

The following is an example execution for the sequential version:

```
ed-vm% ./happyNums
Usgae: ./happyNums -t<1|2|3|4> -lm <base13Number>
ed-vm%
ed-vm% ./happyNums -t 2 -lm 836851
Error, invalid command line options.
ed-vm%
ed-vm% ./happyNums -t2 -lim 836851
Error, invalid limit specifier.
ed-vm%
ed-vm% ./happyNums -ttwo -lm 836851
Error, invalid thread count specifier.
ed-vm%
ed-vm% ./happyNums -t4 -lm 83x6851
Error, limit invalid.
ed-vm%
ed-vm%
ed-vm% ./happyNums -t4 -lm 8396851
------------------------------------------------------------
CS 218 - Assignment #12

Happy/Sad Numbers Program

Hardware Cores: 12
Thread Count: 4
Numbers Limit: 40000000

   Start Counting...
    ...Thread starting...
    ...Thread starting...
    ...Thread starting...
    ...Thread starting...

Results:
--------
Happy Count: 5577647
Sad Count:   34422353
ed-vm%
```