CS 218 – Assignment #12

Purpose: Become more familiar with operating system interaction, threading, and race conditions.
Due: Wednesday (6/27)
Points: 100 70 for program and 30 for write-up
Scoring will include functionality, documentation, and coding style

Assignment:

In recreational mathematics, a Perfect number[1], is a positive integer that is equal to the sum of its proper positive divisors (i.e., the sum of its positive divisors excluding the number itself). For example, the proper divisors of 28 are 1, 2, 4, 7, and 14 which sum to 28. An Abundant number[2], is a number for which the sum of its proper divisors is greater than the number itself. For example, the proper divisors of 12 are 1, 2, 3, 4, and 6 which sum to 16. A Deficient number[3] is a number for which the sum of its proper divisors is less than the number itself. For example, the proper divisors of 21 are 1, 3, and 7 which sum to 11.

The Simpsons, Season 17 Episode 22

The first, 8,191 is a Mersenne prime, a prime number that can be divided only by itself and one. And, 8,128, is a perfect number, where the sum of proper divisors add up to it. Then, 8,208 is a narcissistic number, which has four digits and, if you multiply each one by itself four times, the result adds up to 8,208.

The program should read the thread count (1, 2, 3 or 4) and maximum number limit in binary from the command line in the following format; "./numCounter -th <1|2|3|4> -lm <binaryNumber>".

For example:

```
./numCounter -th 1 -lm 11000011010100000
```

In order to improve performance, the program should use threads to perform computations in parallel.

A main program and a functions template will be provided. All functions must be in a separate source file, **`a12procs.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/Perfect_number
2 For more information, refer to: https://en.wikipedia.org/wiki/Abundant_number
3 For more information, refer to: https://en.wikipedia.org/wiki/Deficient_number

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

- Value returning function ***aBin2int()*** to convert an ASCII/binary string to integer. If the passed string is a valid ASCII/binary value the value should be returned (via reference) and the function should return true. If there is an error, the function should return false. *Note*, the integer being returned is a quad (64-bits) which is different from previous assignments.
- A value returning function, ***getParameters()***, 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 ***aBin2int()*** function.
- A void function, ***numberTypeCounter()***, that will be called as a thread function and determine the count of perfect, abundant, and deficient numbers. The thread must perform updates to the global variables in a critical section to avoid race conditions. See below sections for additional detail.

Thread Function:

Create a thread function, ***numberTypeCounter()*** that performs the following operations:

- Obtain the next number to check (via global variable, ***currentIndex***)
- While the next number is ≤ ***limitValue*** (globally available)
    - If Perfect, atomically increment *perfectCount* variable
    - If Abundant, atomically increment *abundantCount* variable
    - If Deficient, atomically increment *deficientCount* variable

When obtaining the next number to check, the ***currentIndex*** variable, must be performed as a critical section[4] (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 perfect, abundant, or deficient, the appropriate counter should locked (via lock prefix) and incremented accordingly. For example:

```
        lock    inc     qword [perfectCount]
```

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 ***numberTypeCounter()*** 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[5] factor from the base sequential execution and the parallel execution times. Use the following formula to calculate the speed-up factor:

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

4 For more information, refer to: https://en.wikipedia.org/wiki/Critical_section

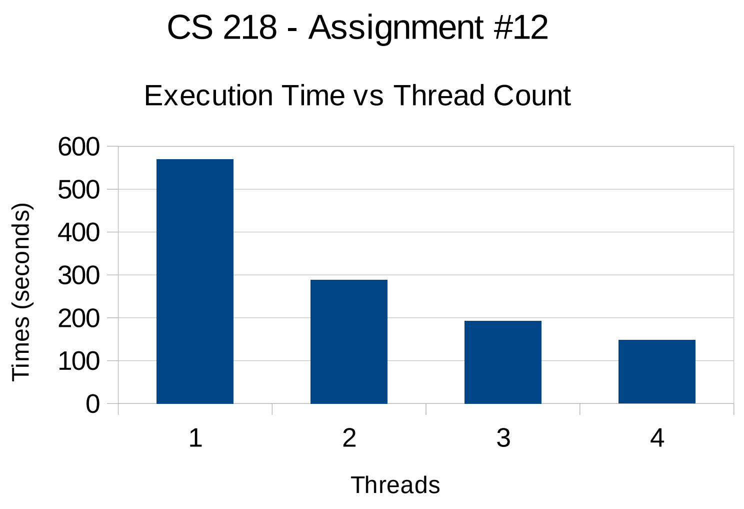
5 For more information, refer to: https://en.wikipedia.org/wiki/Speedup

- 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.

  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 numCounter
```

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.

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 (**`write_up.pdf`**)
        - Includes system description, speed-up amounts, 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 (at a maximum rate of 5 submissions per hour).
- 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	40%	Program must meet the functional requirements as outlined in the assignment.
Timing Script Output	10%	Output from timing script showing results.
Write-Up	40%	Write-up includes required section, percentage change is appropriate for the machine description, and the explanation is complete.

<u>Example Execution:</u>
The following is an example execution for the sequential version:

```
ed-vm% time ./numCounter -th 1 -lm 11000011010100000
------------------------------------------------------------
CS 218 - Assignment #12

Perfect/Abundant/Deficient Numbers Program

Hardware Cores: 12
Thread Count: 1
Numbers Limit: 100000

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

Results:
--------
Perfect Count:   4
Abundant Count:  24795
Deficient Count: 75200

real      0m14.303s
user      0m14.266s
sys       0m0.000s
ed-vm%
ed-vm%
ed-vm%
ed-vm% time ./numCounter -th 4 -lm 11000011010100000
------------------------------------------------------------
CS 218 - Assignment #12

Perfect/Abundant/Deficient Numbers Program

Hardware Cores: 12
Thread Count: 4
Numbers Limit: 100000

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

Results:
--------
Perfect Count:   4
Abundant Count:  24795
Deficient Count: 75200

real      0m3.621s
user      0m14.446s
sys       0m0.008s
ed-vm%
```

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

The following are some addition examples of the applicable error messages.

```
ed-vm%
ed-vm% ./numCounter
Usgae: ./numCounter -th <1|2|3|4> -lm <binaryNumber>
ed-vm%
ed-vm% ./numCounter -th 0 -lm 11000011010100000
Error, invalid thread count value.
ed-vm%
ed-vm% ./numCounter -t 3 -lm 11000011010100000
Error, invalid thread count specifier.
ed-vm%
ed-vm%  ./numCounter -th 4 lm 11000011010100000
Error, invalid limit value specifier.
ed-vm%
ed-vm% ./numCounter -th 4 -lm 11000011010100x00
Error, invalid limit value.
ed-vm%
ed-vm%
ed-vm%
ed-vm%
```