CS 370 – Threading Project

Purpose: Become more familiar with operating system interaction, threading, and race conditions.
Points: 125 (60 for program and 60 for write-up)
Scoring will include functionality, documentation, and coding style

Assignment:

In recreational number theory, a Narcissistic number[1] is a number that is the sum of its own digits each raised to the power of the number of digits. For example, given 8202 which has 4 digits, the sum of each number raised to 4 is 8208.

$$8208 = 8^4 + 2^4 + 0^4 + 8^4$$

Write a C program for Ubuntu (not xv6) to find the count of the Narcissistic numbers between 0 and some user provided limit and display those values. For example, between 0 and 10,000 there are exactly 17 Narcissistic numbers (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407, 1634, 8208, 9474).

The program should read the thread count and number limit from the command line. Refer to the execution examples for information output formatting.

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.

In order to improve performance, the program should use threads to perform computations in parallel. Each thread should obtain a block (parameter initially defined as 1000) of numbers for which to check for Narcissistic numbers. If a Narcissistic number is found, it should be counted and placed in an array. The count, array, and several other variables will be globally defined.

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 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/Narcissistic_number

Command Line Arguments:
The program should read the number of threads and and use the limit from the command line in the following format; "./narNums -th <threadCount> -lm <limitValue>". For example:

```
./narNums -th 6 -lm 10000000
```

If either option is not correct, an applicable error message should be displayed. The allowed range for the thread count is 1 to 30 (inclusive). The minimum acceptable limit value 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. Refer to the C get options example for command line argument handing.

Note, the command line handling **must** be in a dedicated function (not done in the main).

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 simple 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 value 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.

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 100,000,000. The final step will be to chart and explain the results

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.
  - This script will allow unattended execution.
  - *Note*, you may halve the 100,000,000 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:

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

- Remove the locking calls (the 'pthread_mutex_lock' calls and the 'pthread_mutex_unlock') and re-execute the program using a limit of 100,000,000 for 1 thread and then 4 threads.

- Restore the locking calls (the 'pthread_mutex_lock' calls and the 'pthread_mutex_unlock') and change the block size from 1000 to 1. Re-execute the program using a limit of 100,000,000 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:

  - SECTION 1 → Results
    - 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)

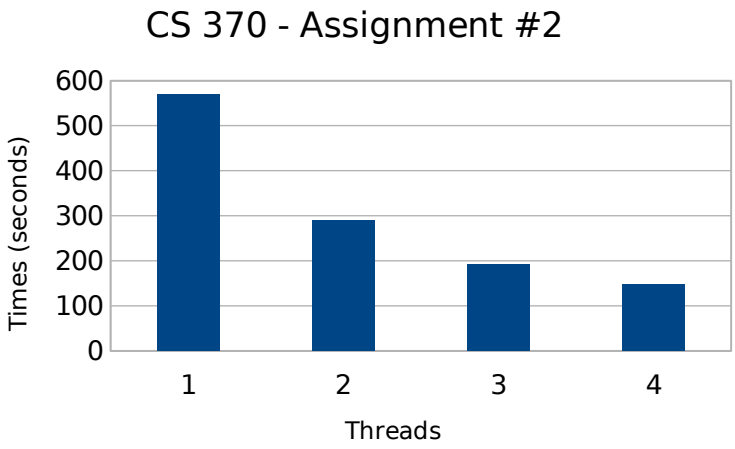
  - SECTION 2 → Locking Calls
    - note the difference in the results with and without the locking calls for the parallel execution
    - explain what specifically what caused the difference (if any)

  - SECTION 3 → Block Size
    - note the difference in the execution times
    - explain what specifically what caused the difference (if any)

  Be sure to label each section appropriately.

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

Below is an example of the time and thread count chart. You can use the provided spreadsheet template or use something else.



The explanation part of the write-up (not including the output and timing data) should be less than ~400 words. Overly long explanations will not be scored.

Submission:

When complete, submit:

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

- 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 22.04 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. *Note*, there are five (5) spaces before each number in the list.

```
ed-vm%
ed-vm% time ./narNums -th 1 -lm 100000000
CS 370 - Project #2
Narcissistic Numbers Program

Hardware Cores: 12
Thread Count:   1
Numbers Limit:  100000000

Please wait. Running...

Narissisic Numbers
     0
     1
     2
     3
     4
     5
     6
     7
     8
     9
     153
     370
     371
     407
     1634
     8208
     9474
     54748
     92727
     93084
     548834
     1741725
     4210818
     9800817
     9926315
     24678050
     24678051
     88593477

Count of Narissistic numbers from 1 to 100000000 is 28

real  0m16.972s
user  0m16.951s
sys   0m0.020s
ed-vm%
ed-vm%
ed-vm%
```

```
ed-vm%
ed-vm%
ed-vm%
ed-vm% time ./narNums -th 4 -lm 100000000
CS 370 - Project #2
Narcissistic Numbers Program

Hardware Cores: 12
Thread Count:   4
Numbers Limit:  100000000

Please wait. Running...

Narissisic Numbers
     0
     1
     2
     3
     4
     5
     6
     7
     8
     9
     153
     370
     371
     407
     1634
     8208
     9474
     54748
     93084
     92727
     548834
     1741725
     4210818
     9800817
     9926315
     24678050
     24678051
     88593477

Count of Narissistic numbers from 1 to 100000000 is 28

real  0m4.274s
user  0m17.093s
sys   0m0.000s
ed-vm%
ed-vm%
ed-vm%
```

```
ed-vm%
ed-vm%
ed-vm%./narNums
Usage: ./narCount -th <threadCount> -lm <limitValue>
ed-vm%
ed-vm%
ed-vm% ./narNums -th 32 -lm 10000
Error, thread count must be >= 1 and <= 30.
ed-vm%
ed-vm% ./narNums -th two -lm 10000
Error, invalid thread count value.
ed-vm%
ed-vm%
ed-vm% ./narNums -th 2 -lm 1g
Error, invalid limit value.
ed-vm%
ed-vm%
ed-vm% ./narNums -th 3 -lm 90
Error, limit must be > 100.
ed-vm%
ed-vm%
ed-vm% ./narNums -t 3 -lm 10000
Error, invalid thread count specifier.
ed-vm%
ed-vm%
ed-vm% ./narNums -th 3 -l 10000
Error, invalid limit value specifier.
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

```
// 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;
}
```

C Get Options Example

```
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<stdlib.h>

/* Global storage for arguments */
struct {
    int threads;
    double limit;
} args;

int
main(int argc, char** argv) {
    char c;
    memset(&args, 0, sizeof(args));
    while ((c = getopt(argc, argv, "l:t:")) != -1) {
      switch (c) {
      case 'l':
          args.limit = atol(optarg);
          break;
      case 't':
          args.threads = atoi(optarg);
          break;
      default:
          fprintf(stderr, "Unknown argument %c\n", c);
          return -1;
      }
    }

    if ((args.threads <= 0) || (args.limit < 100)) {
      fprintf(stderr,
            "Threads (>0 ) and limit (>= 100) arguments required\n");
      return -1;
    }

    printf("Threads:%d Limit:%f\n", args.threads, args.limit);

    return 0;
}
```