CSC 345 – Project #2: Multithreaded Programming

Due: Mar 11, 2022, before midnight

Objective:
Practice multi-threaded programming and understand scheduling policy and locking mechanism.

Due Date:
Mar 11, 2022, 11:55pm

Project Details:
In this project, you are asked to write a multithreaded application that does some laborious tasks simultaneously. Your goal is to develop your application so that it can complete these tasks as efficient as possible. You can assume that you have hardware capable of doing multiple tasks in parallel. Details are described in attached pages, as well as in the textbook.

Program requirements (60 pts):

- [5 pts] The program should take input from **input.txt**, 81 numbers listed up in 9 rows like below:

  6 2 4 5 3 9 1 8 7
  5 1 9 7 2 8 6 3 4
  …(omitted)…
  2 8 5 4 7 3 9 1 6

  Note that the numbers should be **space** separated. Your program should be able to read in this information and print out the read board states correctly. The program also prints out whether the board state is a solution or not. The result should print out in the following format:

  $ ./main 1
  BOARD STATE IN input.txt:
  6 2 4 5 3 9 1 8 7
  …(omitted)…
  2 8 5 4 7 3 9 1 6
  SOLUTION: YES (5.1234 seconds)
  $ _

  No other information should be printed. Of course, if the board state is not a solution, NO should be printed instead of YES. Note that the program takes one input argument "1" from the command line. This is required even if you did not implement the one below.
- [5 pts] Implement another way to setup threads as described in the attached description. That is, rather than creating one thread that checks all nine columns, you could create nine separate threads and have each of them check one column (27 threads). To demonstrate this, your program should use the option "2" instead of "1".
- [20 pts] Devise a statistical experiment that compares the two methods above. Your null hypothesis is "*There is no statistically significant difference between two methods.*" For example, run 50 independent runs, average out the total run time (would be great to print

out the run time, as shown above), and provide an analytical conclusion of your experiments. Graphical plots describing the results would be expected.

- [20 pts] Implement yet another, multi-process way to validate the board. When the option "3" is used, your program should create **three** child processes. Child process 1 will deal with all rows, child process 2 will deal with all columns, and child process 3 will deal with all sub-squares. Repeat the statistical experiments again comparing options "1" and "3." Your null hypothesis is again "*There is no statistically significant difference between two methods.*"
- [10 pts] Repeat the statistical experiments again comparing options "2" and "3." Your null hypothesis is again "*There is no statistically significant difference between two methods.*"

Formatting Requirements

- Add comments to your code (using /* */)
- In a separate, single PDF document, clearly describe which requirements you implemented.
- Your source and executable should be named **main**. Other names will be penalized for formatting violation. When the zip file was extracted, it should look like following:

```
./project2.zip          // your zip submission
./project2              // extracted subdirectory inside your zip file
    |- main.c           // source file
    |- Makefile         // make script (case sensitive!)
    |- input.txt        // input Sudoku board state (81 numbers in 9 rows)
    |- discussion.pdf   // your log with partner
    |- report.pdf       // your description of implementation and
                        experimental results
```

What to turn in

- Zip the folder containing your source file(s) and Makefile, then submit it through Canvas by the deadline. There will be a penalty for not providing any working Makefile.

Task Description Details

Sudoku Solution Validator

A ***Sudoku*** puzzle uses a 9 × 9 grid in which each column and row, as well as each of the nine 3 × 3 subgrids, must contain all of the digits 1 ··· 9. Figure 4.26 presents an example of a valid Sudoku puzzle. This project consists of designing a multithreaded application that determines whether the solution to a Sudoku puzzle is valid.

6	2	4	5	3	9	1	8	7
5	1	9	7	2	8	6	3	4
8	3	7	6	1	4	2	9	5
1	4	3	8	6	5	7	2	9
9	5	8	2	4	7	3	6	1
7	6	2	3	9	1	4	5	8
3	7	1	9	5	6	8	4	2
4	9	6	1	8	2	5	7	3
2	8	5	4	7	3	9	1	6

Figure 4.26 Solution to a 9 × 9 Sudoku puzzle.

There are several different ways of multithreading this application. One suggested strategy is to create threads that check the following criteria:

- One thread to check that each column contains the digits 1 through 9
- One thread to check that each row contains the digits 1 through 9
- Nine threads to check that each of the 3 × 3 sub-grids contains the digits 1 through 9

This would result in a total of **eleven** separate threads for validating a Sudoku puzzle.

However, you can also consider to create even more threads for this project. For example, rather than creating one thread that checks all nine columns, you could create nine separate threads and have each of them check one column. In this case, you will need **twenty seven** threads.

I. Passing Parameters to Each Thread

The parent thread will create the worker threads, passing each worker the location that it must check in the Sudoku grid. This step will require passing several parameters to each thread. The easiest approach is to create a data structure using a struct. For example, a structure to pass the row and column where a thread must begin validating would appear as follows:

```
/* structure for passing data to threads */
typedef struct
{
    int row;
    int column;
} parameters;
```

Both Pthreads and Windows programs will create worker threads using a strategy similar to that shown below:

```
parameters *data = (parameters *) malloc(sizeof(parameters));
data->row = 1;
data->column = 1;
/* Now create the thread passing it data as a parameter */
```

The data pointer will be passed to either the `pthread create()` (Pthreads) function or the `CreateThread()` (Windows) function, which in turn will pass it as a parameter to the function that is to run as a separate thread.

II. Returning Results to the Parent Thread

Each worker thread is assigned the task of determining the validity of a particular region of the Sudoku puzzle. Once a worker has performed this check, it must pass its results back to the parent. One good way to handle this is to create an array of integer values that is visible to each thread. The i^{th} index in this array corresponds to the i^{th} worker thread. If a worker sets its corresponding value to 1, it is indicating that its region of the Sudoku puzzle is valid. A value of 0 indicates otherwise. When all worker threads have completed, the parent thread checks each entry in the result array to determine if the Sudoku puzzle is valid.