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. The figure below presents an example of a valid Sudoku puzzle.

This project consists of designing a multithreaded application that verifies a Sudoku puzzle of any size is valid AND whether a Sudoku puzzle that has 0s instead of numbers can be completed to be a valid puzzle. Completing the puzzle only needs to work on 'easy' puzzles.

Important: The starter code defines a 2D array called grid where row-0 and column-0 is ignored. This makes it easier think about the puzzle. Fro the puzzle below

- grid[1][1] = 6
- grid[1][9] = 7
- grid[9][1] = 2
- grid[9][9] = 6

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

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

- A thread to check that each column contains the digits 1 through 9
- A thread to check that each row contains the digits 1 through 9
- Nine threads to check that each of the 3 × 3 subgrids contains the digits 1 through 9

This would result in a total of eleven separate threads for validating a Sudoku puzzle. However, you are welcome 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.

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;
```

Pthreads could 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, which in turn will pass it as a parameter to the function that is to run as a separate thread.

Your program will also need to use a struct, but it is up to you what is inside the struct.

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 or possibly finding if there is a single 0 in the region

which could be converted to a number. Once a worker has performed this check, it must pass its results back to the parent.

There are multiple ways to achieve this. The parameter that is passed into the thread could have a result component where all the different threads can access. The *ith* index in this result array could correspond to the *ith* worker thread. The worked can use different numbers to indicate its status, say 1 for valid, 0 for invalid, 2 for incomplete, etc.

When all worker threads have completed, the parent thread checks each entry in the result array to determine if the Sudoku puzzle is valid.

Extension

Your program should be able to handle not just 9x9 Sudoku puzzles, but any size Sudoku puzzle. See the starter code on how to take an argument from the command line, indicating the filename for the puzzle, and how to read puzzle.

The format of the puzzle is as follows. A number indicating the Size x Size of the Sudoku puzzle followed by Size x Size of integers.

```
4
3 4 2 1
2 1 3 4
1 3 4 2
4 2 1 3
```

The valid numbers for a Sudoku puzzle are 1 to Size. If the puzzle has a zero, that indicate that the puzzle is not yet complete and your program should try to fill it with the correct number.

For example, for the puzzle below.

```
3 4 2 1
2 1 0 0
1 3 0 2
4 2 1 3
```

The location grid[3][2] can be filled with "4" since row-3 already has 1, 2, 3 but not 4. Once that is filled, grid[2][3] can be filled with "3" since column-3 has 2, 4, and 1 but not 3. Your Sudoku solver should be able to fill cases of 1-missing-number, but does not need to be any more clever than that.

What to Submit

- Write a multi-threaded program using the starter code that use pthreads that
    - Verifies if a given Sudoku puzzle is valid

        - Completes a Sudoku puzzle if not valid.
    - If the puzzle is complete, your program should print:
        - "Complete puzzle? true"
    - Only for complete puzzles, or puzzles that your program was able to complete, print
        - "Valid puzzle? true"
    - Substitute "false" for the above statements is puzzle is not complete or not valid.
    - You can, but do not have to use the starter code, but your program must be runnable from command line as follows

```
./sudoku some-puzzle-file.txt
```

    - **sudoku.c:** Your solution (and optional sudoku.h). Include instructions on how to compile and run, such as:
      `// compile: gcc -Wall -Wextra -pthread -lm -std=c99 sudoku.c -o sudoku`
      `// run (verify): ./sudoku puzzle-easy.txt`
      `// run (hard): ./sudoku puzzle-hard.txt`
    - **self-assessment.txt:** Indicate what you have and have not implemented, such as:
        - Can verify whether it is valid or not - /10
        - Starts/joins multiple threads to speedup operation - /5
        - For NxN complete puzzle, can verify whether it is valid or not - /5
        - The number of threads is proportional to N - /5
            - Always using 3 threads, no more no less.
        - **Bonus**: Can complete puzzles - /2
        - **Bonus:** Can complete difficult puzzles - /3

```
4

3 0 0 0

2 1 0 0

0 0 0 0

4 2 1 0
```

        - Total: 25 (+ Bonus 5)

      Submit: **sudoku.c**, **self-assessment.txt** based on the rubric below and **some of your sudoku test puzzles**.

Tips

- The puzzle will always be a perfect square 2x2 (size-4), 3x3 (size-9), 4x4 (size 16) etc

- Checking for **complete** is easy. If there are no 0s in the puzzle, the puzzle is complete.
- Checking for **valid** requires checking each row, column and box to make sure there are no repeated numbers. "valid" is only meaningful for complete puzzles. For puzzles that are incomplete, we do not have a a notion of valid or invalid.
- If you are not doing the bonus part, that is you are **not completing the puzzle** (i.e. not replacing any 0s with actual numbers), you can check if it is complete and then if necessary check whether it is valid.
- If you are doing the bonus part, that is you are **completing the puzzle** (i.e. replacing any 0s with actual numbers), you should try to replace as many 0s as possible. When no more 0s can be replaced, you can check for complete and then validity for any possible duplicates. Replacing 0s with actual numbers will be similar to the checking validity, so you can simplify your code by combining those steps.
-