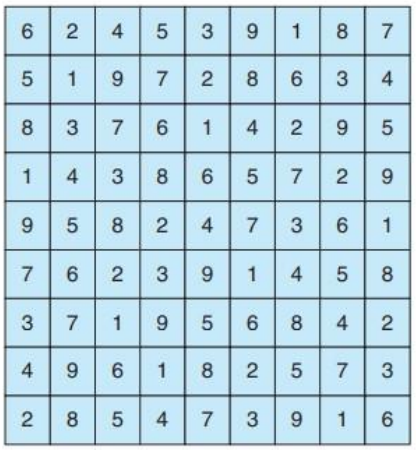
Semester-Project

Phase I—Sudoku Solution Validator



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

A Sudoku puzzle uses a 9 × 9 grid in which each column and row, as well as each of the nine 3 × 3 sub grids, must contain all the digits 1 ··· 9. Figure 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. 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 sub grids contains the digits 1 through 9 columns

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

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. The *ith* index in this array corresponds to the *ith* 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 would indicate 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.

Conditions:

- Threads are not allowed to update other part of the array. Other threads can only read the value from the array. (For example, thread ***i*** can only write on ***ith*** location, for other parts it can only read from the array.)
- When thread returns values to Main thread and it is writing on array, no other thread can read array value.
- Thread will return the value to Main thread.
- Here you must synchronize the task and apply mutex lock.
- Thread can cancel the other thread, if found an invalid entry. But you must set the criteria for cancelation. (For which thread you need to set accept cancelation or not.)
- Storing the invalid entries and their indexes.

EXPECTED OUTPUT: Phase I

Main thread will display the final result

- Invalid boxes in the puzzle (Values and indexes)
- Invalid rows (Values and indexes)
- Invalid columns (Values and indexes)
- Thread ID with each invalid output showing which thread has calculated the result.
- Cancelled thread id (both threads IDs)
- Comparison results of invalidation. (e.g., Row one has repeated value 3).
- Total count of invalid entries.

- Output statement showing that invalidation is because of invalid digits (less than 0 or greater than 9) or numbers are placed at invalid locations.

6	2	4	5	3	4	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

Phase 2—Sudoku Puzzle Solution

Phase 1 of this project is about the validation performed on the matrix (9 x 9). In phase 2, you must initialize a matrix with at least 2 invalid entries and provide its solution. You can store the matrix in a file. Each thread is assigned to each row and column and change the corresponding values if found erroneous. (Same as followed in phase 1). Solution of the sudoku puzzle, threads attributes must be initialized in start. So that Thread cancelation can be achieved. (As in solution thread has to swap the values
with other threads, you need to keep track of changes. If a thread changes another threads value, thread don't allow to change its value it will cancel the other thread.)
When a thread wants to swap the position of digits, it sends the signal to corresponding thread (Optional). Each thread must have a criterion to allow a thread to swap its position with other threads. Limited threads will be running using semaphores to balance the overhead and security issues, e.g., while updating the matrix no other thread will read the data. Threads should be created efficiently. After resolving the invalid entries, you must have to validate the solution and display the result.

EXPECTED OUTPUT: Phase II

Main thread will display the following results:

- Original and Resultant Puzzle in Matrix format.
- Total invalid entries.
- Threads created against each invalid entry.
- No of moves for each invalid entry.
- IDs of all threads with their ***ith*** indexes.
- Location of invalid entry and its new position.

Project Report Requirements

- Provide Pseudo codes for Phase I and II.
- Appropriate illustrations of the operating system concepts you used in your Pseudo codes.
- Provide your implemented codes.
- Provide your system specifications.
- Short paragraph how could you implement these concepts in some other scenario. (at least one)

Other Instructions:

Use any appropriate and efficient synchronization technique for this project. The code must be properly commented, and this carries marks too. Group details should appear on separate page in report, and you should also mention the exact contribution of each group member to the project. Again, all this carries marks.

Do test your program thoroughly and make sure it gives correct output before submitting. References should be relative and not absolute and to crosscheck you can just copy your executable to some other machine to see if it works there as well. If your program does not compile or works, you do not get credit. You will not be allowed to debug your program during the demo. We will not entertain excuses like “it was working before, but we don’t know what happened to it now!!!”