Introduction

The following report is written for assignment 1 of COMP2006 Operating Systems. This report is written by Kevin Kongo, 21473719. Please find the Declaration of Originality included in the zip file with this report. The program “mssv” is a multi-threaded sudoku solution validator written in C. A parent thread creates 4 threads that are each tasked with validating separate portions of a completed sudoku board, to ensure the board is legal.

The following is a snapshot from the README.txt file found in the source code folder for this assignment:

to compile the program: in Linux terminal, open the path to the code. then run the command:

make

This will execute the makefile and compile the code. Ensure GCC is installed.
To run the program:

mssv solution delay

Where solution is the file that contains the Sudoku Solution to be validated, and delay is an integer with a value between 1 and 10.

Three solution files are provided: solution.txt, solution1.txt, and solution2.txt. The first one is a valid solution while the other two are not.

This assignment will discuss the methods for achieving synchronization where synchronization was necessary, the threading implementation that was use, and a description of how the program was tested to ensure correct functionality.

Synchronization

```
typedef struct shared{
    int* col;       // all of the shared data
    int* row;
    int* sub;
    int** board;

    int* counter;
    pthread_mutex_t* mutex_c; // mutex for count
    int* total;
    pthread_mutex_t* mutex_t; // mutex for total checks that are done on sudoku board

    pthread_cond_t* wake_cond; // used for the last thread to wake the parent process
```

As can be seen in this struct, taken from threading.h in the source folder for this assignment, there are a total of six shared variables. Notice how all values in the struct are pointers, emphasizing the point that

the threads are all accessing data that is shared. Five of these values are outlined in the assignment task sheet. Those are:

Counter: counter is an integer that is used to track the total number of correct validations on a given sudoku board, and is incremented after every successful validation of a row, column, or sub-grid. After the threads finish executing, the value of counter is checked by the parent thread and then output to the terminal, indicating whether or not the sudoku board as a whole is a valid solution. This is only the case if counter is 27. Any less, and the board is invalid. Counter is synchronized using the mutex lock mutex_c. This is because incrementing a value in memory consists of multiple CPU instructions, any of which can be pre-empted by another thread. This makes incrementing counter a race condition, and hence it is synchronized. In the program, the mutex_c mutex is not locked when the final check is done by the parent, as it is known at this point that all child threads have finished executing and can no longer cause a race condition on the counter.

Board: This references the 2D array storing the solution that was imported from an input file. It is only ever read by the threads, never written to. This means there is no race condition caused when accessing the board array concurrently, therefore not requiring synchronization.

Col, Row, and Sub: These three pointers point to an array of length nine. They contain integers that indicate whether each index of a row, column, or subgrid is valid. As required in the assignment specification, these arrays are expected to be concurrently accessed, as multiple threads update values in the relevant array to indicate the presence of an invalid row, column, or subgrid. However, no synchronization methods are required here. This is because as per the assignment specification, each thread is allocated a separate portion of the three arrays. For example, only thread four accesses the col array, thread one only accesses row[0], row[1], and row[2], while thread two only accesses row[3], row[4], and row[5], etc. This means that the threads will never have a race condition on the three arrays, as they never are required to make adjustments to the same location in memory within the arrays.

Along with these give values, another counter was implemented along with a condition variable. They can be seen in the variables total and wake_cond. Like counter, total is incremented after every validation, however the difference is that total is incremented after every validation, regardless of whether or not the validation was successful. This is used along with wake_cond. The Pthread Library includes functions that block threads using condition variables. The parent thread of this program calls pthread_cond_wait() after initializing and creating each child thread. This is as required in the assignment specification. Afterwards, each thread will lock the mutex_t variable associated with total after every validation, then increment total by 1. This is done until the last thread finishes its validation, and total reaches 27. When this is the case, the thread that incremented total last will print to the terminal indicating that it was the last thread to complete a validation, and then it will wake the parent thread, using the wake_cond variable and calling pthread_cond_signal(). The parent thread will then join into each thread to clear the threads from memory, and then print the final results of the program. Without a second counter variable, it is difficult to track which thread was the last to complete a validation. Counter can not be used for this purpose, because counter is not guaranteed to reach 27 or any specific value before runtime.

There was no other synchronization required for this assignment.

Testing procedure

As the student who wrote the program independently, I believe it to be functional in any test case that it is required to complete. To ensure that all synchronization, threading, and validation logic was implemented correctly, the following methods were used:

Printf(): Print statements would be used at several points within the program. This is to ensure that inputs are being read correctly, as well as observing the change of certain shared variables within the program. For example, this method was used to track the previously mentioned counter and total shared variables to ensure they were finishing on the values that were expected. It was also used to observe the concurrency of each thread, as mentioned in the next point.

Sleep(): each thread would be put to sleep for a given amount of time using the delay argument when executing the program. Printf() statements would be used just before each sleep() function was called to identify the point at which each thread went to sleep. As expected, 4 prints are made to the terminal, virtually at the same time, then the input time in seconds passes before another 4 prints are made. This is to be expected in a multithreaded environment, as each thread sleeps independently from one another. If threading was not implemented correctly, this behaviour would not be observed. Furthermore, sleep() was used to identify issues with synchronization that would be caused by incorrect implementation of condition variables and mutex locks. As outlined in the assignment guidelines, this is used to observe synchronization errors, with the expectation that the output of the program may change, given a delay of between zero seconds and at least 1 second for every validation. In my program, no differences were observed.

Test cases: multiple test files with different solutions were used as inputs for the program as well. This is used to test the logic that ensures validations are made correctly and to prevent any case-by-case correct outputs that could have been caused by using hard-coded values. In the case of my project, this helped to identify an issue that meant the program incorrectly validated the solution, which meant it identified rows/columns/sub-grids that had “correct” values within them, when this was not the case. The problem was addressed, and the program now validates the board correctly.

Test cases

Firstly, the program assumes that the test cases (solution text files) follow a consistent and correct layout and ordering. It also assumes that each value is an integer from 1-9. There is error checking for the file, however there is no checks to ensure that each value in the text file is an integer from 1-9.

As seen in the source folder, there are a total of three test cases.

solution.txt is a solution with all valid solutions. When executing the program with the following arguments:

./mssv solution.txt 1

The output is expected to be correct, with it describing all 4 threads having correct validations. The output is as follows, and all inputs/outputs can also be found in the sampleoutput.txt file as well:

Thread ID-4 is the last thread.

Thread 1 ID-139647312418368: valid

Thread 2 ID-139647304025664: valid

Thread 3 ID-139647295632960: valid

Thread 4 ID-4294967296: valid

There are 27 valid rows, columns, and sub-grids, thus the solution is valid.

solution1.txt and solution2.txt are solutions that are incorrect. Only a few rows, columns, and subgrids are correct.

For solution1.txt, row 1 and subgrid 9 are the only ones that are valid. With the following input:

./mssv solution1.txt 1

The output shall indicate all incorrect rows, subgrids, and columns, with the only correct ones being the previously mentioned two. The output is as follows, and is correct:

Thread ID-4 is the last thread.

Thread 1 ID-140713752282688: Sub-Grid 1, Row 2, Sub-Grid 2, Row 3, Sub-Grid 3, is/are invalid

Thread 2 ID-140713743889984: Row 4, Sub-Grid 4, Row 5, Sub-Grid 5, Row 6, Sub-Grid 6, is/are invalid

Thread 3 ID-140713735497280: Row 7, Row 8, Sub-Grid 8, Row 9, Sub-Grid 9, is/are invalid

Thread 4 ID-4294967296: column 1, column 2, column 3, column 4, column 5, column 6, column 7, column 8, column 9, is/are invalid

There are 2 valid rows, columns, and sub-grids, thus the solution is invalid.

Similarly, using solution2.txt as the input also provides the correct output. Sub-grid 4, row 5, and Column 9 are the only valid sections of the solution, and using the following input:

./mssv solution2.txt 1

The output is as follows:

Thread ID-4 is the last thread.

Thread 1 ID-140333189776960: Row 1, Sub-Grid 1, Row 2, Sub-Grid 2, Row 3, Sub-Grid 3, is/are invalid

Thread 2 ID-140333181384256: Row 4, Sub-Grid 5, Row 6, Sub-Grid 6, is/are invalid

Thread 3 ID-140333172991552: Row 7, Sub-Grid 7, Row 8, Sub-Grid 8, Row 9, Sub-Grid 9, is/are invalid

Thread 4 ID-4294967296: column 1, column 2, column 3, column 4, column 5, column 6, column 7, column 8, is/are invalid

There are 3 valid rows, columns, and sub-grids, thus the solution is invalid.