Jackie Diep

Professor Bo Li

CSC 306 G001

29 November 2020

Programming Assignment: Multi-Threaded Programming

Abstract: Multi-Threaded programming is important because it allows the processor to perform multiple tasks concurrently. As we reach the limits of singular processors, multi-threaded programming is one of the best ways to optimize performance, and it is expected to remain one of the most important expansions to development for the foreseeable future. This paper will detail a basic implementation of multithreading in validating the answer of a Sudoku puzzle.

Program Description

For the Multi-Threaded Programming Assignment, I chose the Sudoku Solution Validator because it was an interesting real-world example. The assignment has an expected implementation of eleven threads, though more could easily be implemented. Nine threads are to be dedicated towards validating the nine 3 x 3 Squares. The remaining two threads are dedicated to Row and Column validation. In addition to the thread dedications, the assignment also recommends two sections of code: a struct for parameter passing and the preliminary initializations before thread creation.

Initial Programming

The initial aspects of the assignment were relatively easy and very quick to implement. This includes initializing the given example Sudoku matrix into a 2d array and creating functions to validate individual pieces correctly. The main difficulties and learning experiences of the assignment derive from the implementation of the threads and passing the struct parameters into them. I ran into many errors of “Invalid data type <struct parameter>” and dereferencing issues that I fixed by changing the implementation of the struct. The assignment recommends to use typedef and an unnamed struct.

```
typedef struct
{
        int row;
        int column;
}parameters;
```

I am not particularly familiar with implementing this type of struct and adding pointers to the implementation led to a long sequence of troubleshooting. Instead of using the assignment’s recommendation, I used a normal struct.

```
struct parameters
{
        int row;
        int column;
        int matrix[9][9];
};
```

Also note that I added the int matrix[][] 2d array to the struct definition to hold the Sudoku matrix.

My implementation of the validation themselves was nested for loops that iterated through their relevant sections that broke and returned a value if a duplicate was found. The Row and Column validator implementations were very straight forward, though some problems were had with implementing the threads. The 3 x 3 Square validator ran into many more complicated issues that I managed to resolve eventually. The memcpy() function was used to copy the initial Sudoku matrix to the structs, and a simple for loop was used to assign the respective variables and call for thread creation.

Thread Implementation

The eleven thread Ids were initialized using eleven simple pthread_t tid lines, and it quickly became clear that I would have to use a thread Id pointer array for any type of code elegance later on. I used the assignment's suggestion of the struct data pointer call, but I also added my own struct array pointer that I will speak more on shortly.

```
struct parameters* data = (struct parameters*)malloc(sizeof(struct parameters));
```

Row and Column Thread Implementation

Once I implemented a basic struct instead of the assignment's recommendation, the implementation of the Row and Column threads was very straight forward. Before I did, I ran into many memory and incorrectly initialized errors due to unfamiliarity.

The Row and Column threads are threads number 10 and 11 in my code respectively, and their implementation looks like:

```
else if (i == 9)
{
        data → row = 0;
        data → column = 0;
        pthread_create(pthreads[i], NULL, rowThread, (void*) data);
}
else if( i == 10)
{
        data → row = 0;
        data → column = 0;
        pthread_create(pthreads[i], NULL, columnThread, (void*) data);
}
```

Inside of the functions, another parameter pointer struct is created and set equal to the passed in data struct such that the variables can be accessed. To correct the biggest issue I had while implementing the threads in this assignment, I also initialized local variables equal to the dereferenced variables of the new struct pointer. This way, the Row and Column threads would not overlap and change variables before they could be properly used by one another.

```
struct parameters*rowData = (struct parameters*) data;
int row = rowData → row;
int column = rowData → column;
```

Following this, the actual validation of the Rows and Columns is simply two nested for loops with an initialized local array of nine elements to check for repeated values per Row or Column. As new values are found, those values are added to their respective place in the array. For example, a value of 9 on the matrix would be placed into array element number 9 (aka. index number 8). If an array element was found to not be equal to zero when the value was found, it would be considered a repeat, and the thread function would end.

```
for (row or column)
{       initialize array to hold repeats
        set row or column equal to zero (whichever is nested for loop)
        nested for loop (row or column)
        {
                if ( duplicate) { return invalid }
                else { array[#] = # }
        }
}
return valid
```

3 x 3 Square Thread Implementation

The 3 x 3 square threads were not nearly as simple to implement as the Row and Column threads. I ran into many overlap issues where the rows and columns would constantly be changed before they could be used correctly. I assume that the problem could easily be solved with numerous struct declarations for parameter passing such that overlap would be a nonissue, but I was not willing to implement such an inelegant solution. Eventually, I decided on using an array of struct pointers that would index with the loop that I was already using to call for thread creation.

```
struct parameters* squareData[9];
for (int i = 0; i < 9; i++) { squareData[i] = malloc(sizeof(struct parameters)); }
```

This solution ended up working better than intended and immediately fixed all of my remaining issues with the implementation of the 3 x 3 Square thread implementation.

The for loop I used to call for thread creation also handled starting point allocation of the threads.

```
for ( index threads )
{
      if ( 3 x 3 square thread)
      {
               if ( first three threads )
               {
                          First row of squares
               }
                else if ( next three threads )
                {
                          Second row of squares
                }
                else if ( last three square threads )
                {
                          Third row of squares
                 }

         call memcpy() for each struct in the array as the loop iterates

         // Calculate the appropriate column to begin per thread
         squareData[i]->column = i % 3 * 3;

         pthread_create(pthreads[i], NULL, squareThread, (void*)squareData[i]);
      }
}
```

The last section of the 3 x 3 Square implementation is the thread function used for it. The initial parts involving another struct and variable initializations are the same as the Row and Column threads, but the square thread implements two “bound” variables because the squares are 3 x 3 instead of from end to end of the matrix.

```
int rowBound = row + 3;
int columnBound = column + 3;
```

The array to hold repeats is also initialized outside of the for loops for the square thread because the entire 3 x 3 structure is compared, instead of individual rows or columns. The validation for loops for the squares is the same as the Row and Column ones, with the exception of using the “bound” variables instead of 9 as the end of the loops or 0 as the reset value of the loops.

Result Passback

The final section of my report, Result Passback, was simple to implement once the issues with Square threads were solved. I initialized an array to hold returns from the threads as suggested by the assignment, and I initialized a pointer variable to capture returns and add them to the array. Each thread function initializes and maintains a pointer equal to one, and if an invalid input is found in the matrix, the pointer is set equal to zero and the function returns the pointer. If all inputs are valid, the pointer is returned as one.

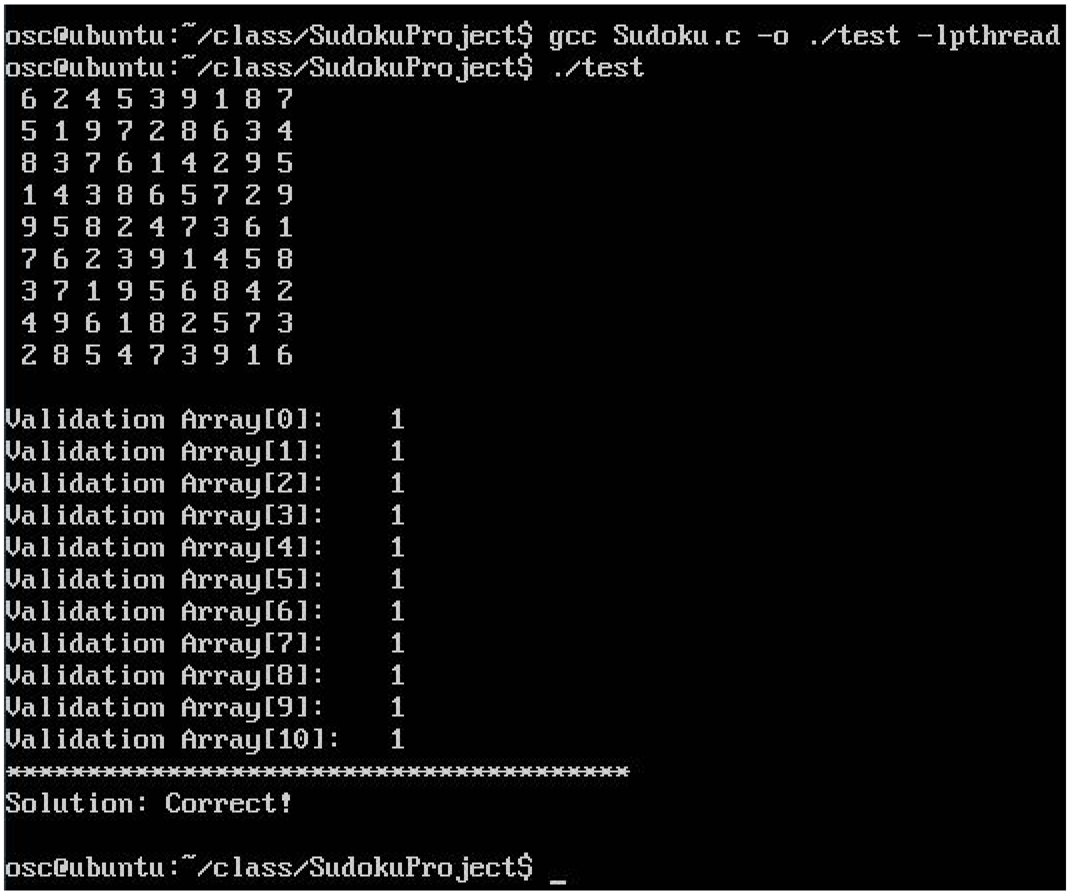
```
        int * validator = (int *)malloc(sizeof(int));
        *validator = 1;
        if ( invalid result )
        {
                *validator = 0;
                return validator;
        }
if all results are valid
return validator;
```

Back inside of the main function, threads are allowed to complete and thread returns are captured using a for loop.

```
        for ( thread index )
        {
        pthread_join(*pthreads[i], (void*)&ptr);
        validationArray[i] = *ptr;
        }
```

Finally, the validation array is processed, and if any array elements are equal to zero, main outputs an invalid response. If all array elements are one, a valid response is given. Once the malloc()'d array elements and parameter struct are freed, the program exits.

A Correct Solution:

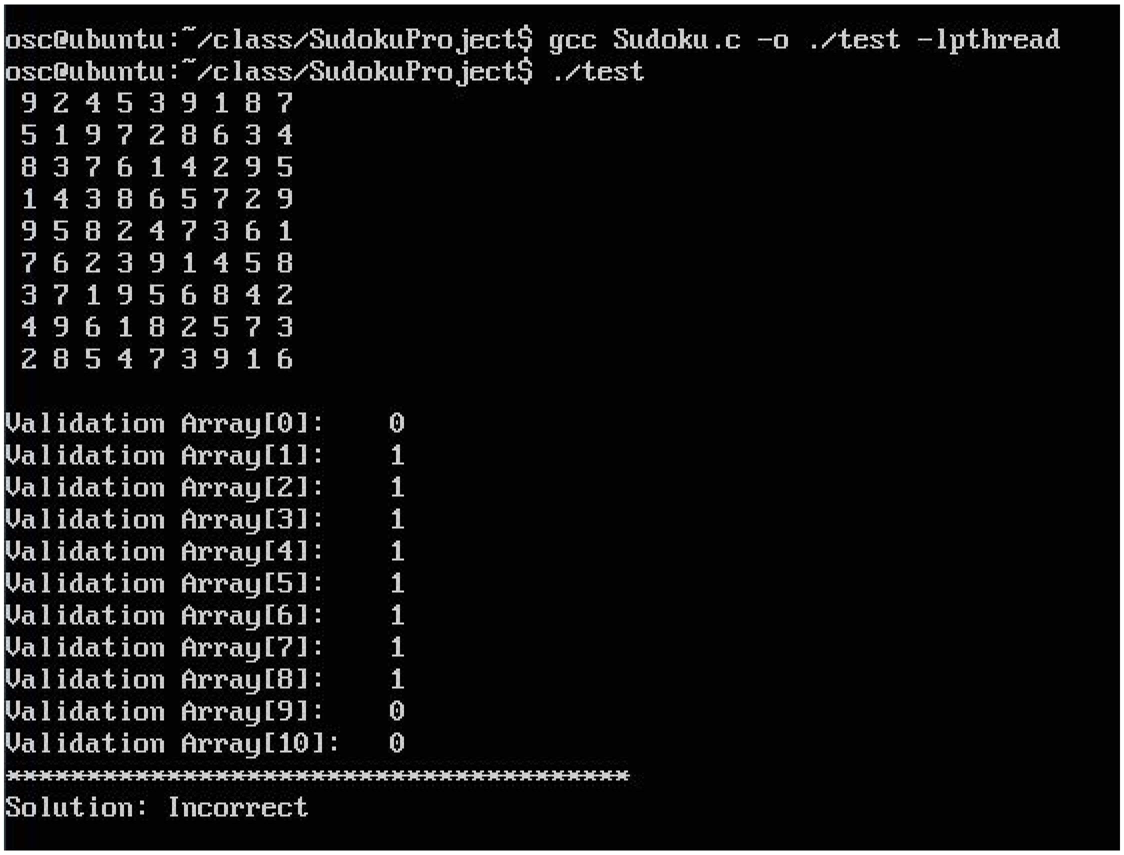


```
osc@ubuntu:~/class/SudokuProject$ gcc Sudoku.c -o ./test -lpthread
osc@ubuntu:~/class/SudokuProject$ ./test
 6 2 4 5 3 9 1 8 7
 5 1 9 7 2 8 6 3 4
 8 3 7 6 1 4 2 9 5
 1 4 3 8 6 5 7 2 9
 9 5 8 2 4 7 3 6 1
 7 6 2 3 9 1 4 5 8
 3 7 1 9 5 6 8 4 2
 4 9 6 1 8 2 5 7 3
 2 8 5 4 7 3 9 1 6

Validation Array[0]:    1
Validation Array[1]:    1
Validation Array[2]:    1
Validation Array[3]:    1
Validation Array[4]:    1
Validation Array[5]:    1
Validation Array[6]:    1
Validation Array[7]:    1
Validation Array[8]:    1
Validation Array[9]:    1
Validation Array[10]:   1
***************************************
Solution: Correct!

osc@ubuntu:~/class/SudokuProject$ _
```

An Incorrect Solution:



```
osc@ubuntu:~/class/SudokuProject$ gcc Sudoku.c -o ./test -lpthread
osc@ubuntu:~/class/SudokuProject$ ./test
 9 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

Validation Array[0]:    0
Validation Array[1]:    1
Validation Array[2]:    1
Validation Array[3]:    1
Validation Array[4]:    1
Validation Array[5]:    1
Validation Array[6]:    1
Validation Array[7]:    1
Validation Array[8]:    1
Validation Array[9]:    0
Validation Array[10]:   0
***************************************
Solution: Incorrect
```

The top left matrix variable was changed into a 9. The first 3 x 3 Square validator, thread 1, is invalid, as well as the Row and Column validators, threads 10 and 11 respectively.

References

1. C structs and Pointers. (n.d.). Retrieved from https://www.programiz.com/c-programming/c-structures-pointers. Accessed November 28, 2020.

2. “pthread_create”. The Open Group Base Specifications Issue 7, 2018 edition. (n.d.). Retrieved from https://pubs.opengroup.org/onlinepubs/007908775/xsh/pthread_create.html. Accessed November 28, 2020.

3. “pthread_join”. The Open Group Base Specifications Issue 7, 2018 edition. (n.d.). Retrieved from https://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_join.html Accessed November 28, 2020.