CSCI 3020U

Lab #3: Sudoku Solution Validator

Alexandar Mihaylov 100536396

Jeremy Kwok - 100341977

Taylor Smith - 100372402

Elias Amal - 100494613

Luisa Rojas - 100518772

Objective

Design a multithreaded application that determines whether the solution to a sudoku puzzle is valid.

Program Flow

Passing Parameters to Each Thread:

Our implementation first opens a file **puzzle.txt** and loads the Sudoku into a 2D array defined within the structure:

```
typedef struct{

    int sudokuGrid[9][9];
    int Rselector;
    int Cselector;
    int Bselector;

} parameters;
```

The program starts iterating through the first column and finds the first empty(zero) slot of the Sudoku using the `solve(int row, int col, parameters *data)` function.

If the current square being validated is **not** 0, then it skips over it and increases the column number by one. Then, if the current column is 9, increase the row number by one and set the column number to 0. After this, check if the current row number is 9; if it is, then the validation is done and the return value is `true`.

We then create three different threads (using a `for` loop to iterate through the possibilities), one for validating the box, one for validating the column, and lastly one for validating the row:

```
pthread_t rowthread;
pthread_t colthread;
pthread_t boxthread;

data -> Bselector = (col/3) + (row/3)*3;
pthread_create(&boxthread, 0, validate_box, (void *) data);

data -> Cselector = col;
pthread_create(&colthread, 0, validate_col, (void *) data);

data -> Rselector = row;
pthread_create(&rowthread, 0, validate_row, (void *) data);
```

As seen above, the data pointer will be passed to the `pthread_create()` function, which in turn will pass it as parameter to the funtion that is to run as a separate thread.

Returning Results to the Parent Thread:

When a worker (thread) finishes it's valitidy check, it passes its results back to the parent using the `pthread_join()`

function, passing its corresponding thread and validation boolean value as parameters.

```
(void) pthread_join(boxthread,&boxvalid);
(void) pthread_join(colthread,&colvalid);
(void) pthread_join(rowthread,&rowvalid);
```

Once we join the validating threads, the parent checks that the box, column, and row validators return true. If they do, then they are called recursively to solve the next row,column iteration. The function eventually terminates when the row reaches 9 and it returns 1.

In the event that we have iterated through all the possibilities and no value was set, then we must set the current `row,col` to 0 and return `false`.

```
data -> sudokuGrid[row][col] = 0;
```

Reading the Puzzle and Writing the Solution:

Note that the content of the **puzzle.txt** file is the Sudoku puzzle to solve, using spaces to delimit each column. For this reason a tokenizer had to be implemented: `tokenize2(char *input, char *delim)` when loading the Sudoku Grid from the file.

```
char buffer[BUFFER_LEN];

for (int i = 0; i < 9; i++){
fgets(buffer,BUFFER_LEN, f);
char **out = tokenize2(buffer, " ");
for (int j = 0; j < 9; j++){
  data->sudokuGrid[i][j] = atoi(out[j]);
}
```

After it's been established by the `solve(int row, int col, parameters *data)` function that the puzzle is solvable, the solution to said puzzle is written to a file called **solution.txt** in the same format as the input file and the array is printed on the screen.

```
FILE *file = fopen("solution.txt", "w");
```

Example of `solution.txt`

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