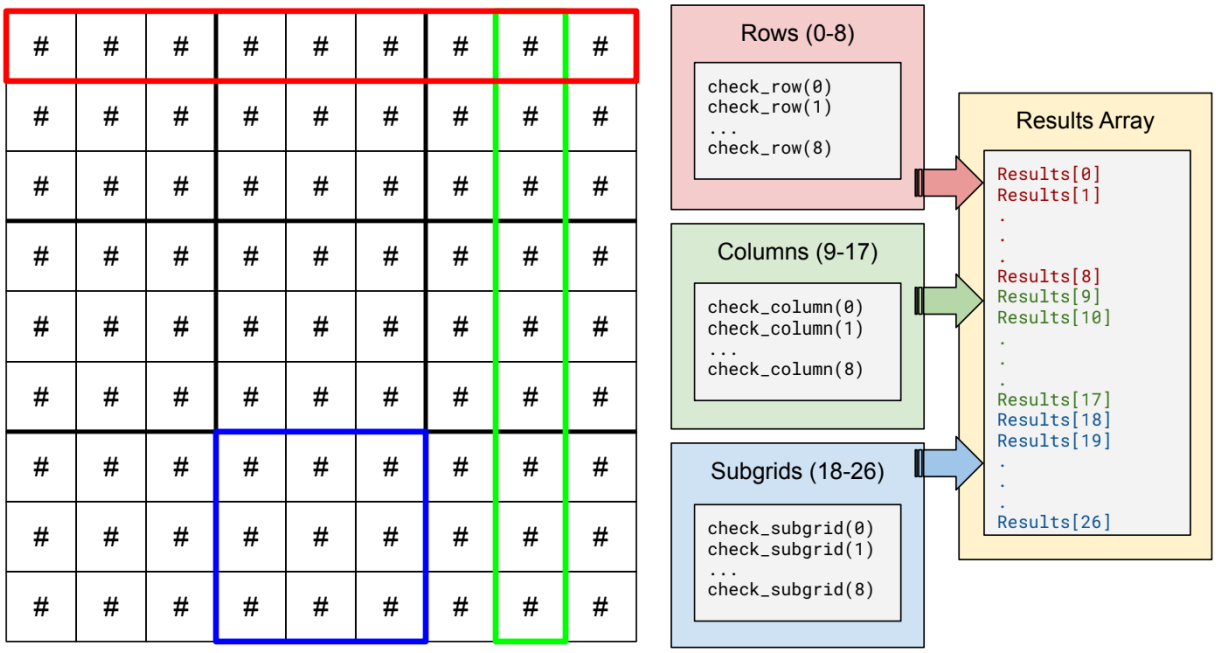
Austin Howard
34980586
2025-06-28
Lab 1 Report
Sudoku Solution Validator

Solution Design [15 points]

Instructions: Describe the design of your solution. You may also include a diagram/diagrams.

My program (see source code in this report's directory) validates an already completed 9x9 sudoku puzzle using multithreading. As instructed, it is using POSIX's PThread library. This solution divides the validation process into twenty-seven different threads – nine rows, nine columns, and nine 3x3 subgrids, which we will refer to here as *elements*. Each thread independently checks that its assigned element contains one and only one occurrence of each digit (1 to 9).



For this design, an already completed sudoku puzzle is supplied with a known valid solution. This is useful as a test case, as all that is needed to make it invalid is to change one digit.

To point out specific coordinates on the grid, the `parameters` struct is used. Its arguments are dynamically allocated to prevent race conditions.

Each validation function dynamically allocates an integer (`1` being valid, `0` being invalid) `result` which is returned through `pthread_exit()`. This is stored in the global `results[]` array, which consists of the 27 elements (rows, columns, and subgrids) used to validate a sudoku grid. The result of each thread is dynamically allocated and freed after being stored.

After joining all the threads, the `results[]` array is then analyzed by the main function to determine if any of the threads returned as invalid (`0`). If no threads return invalid than the sudoku solution provided is considered valid.

Parallelism significantly reduces the time it would take to run this program on multi-core systems; it's among the primary benefits of using this design. My use of modular thread functions help improve the readability of the code, which helps facilitate testing its design. Output includes helpful features such as a visual representation of the puzzle as well a debugging feature which tracks each `result` in `results[]`.

Challenges and Solutions [15 points]

Instructions: Explain some of the challenges you experienced during the implementation of your solution and how you addressed them.

Challenge 1: The first challenge was presented upon the realization that my programming environment was not well suited for this project. Typically, I code with CLion on my Windows 11 laptop. As a security researcher, I find that the Windows library is useful for practical offensive/defensive tactics. I am also a heavy user of Python and really enjoy PyCharm, so I tend to stick with JetBrains' IDEs. However, with the stipulation that PThreads must be used, I could not use this setup for the project. Attempts at getting PThreads to work on Windows (using MSYS2 to set up a UNIX-like environment) were scrapped due to complexity and lack of interest in diluting my personal workstation environment.

> **Solution 1:** After a lot of debate, a Kali Linux VMware environment was utilized for this project. I already use it for research, and it is a POSIX friendly environment. However, since the VM could only access limited hardware resources, careful consideration was given for a lightweight IDE that still had full features for syntax checking and GitHub integration. VS Code was selected because of its modularity, support for C, and full feature set. Although, this solution did still frequently bog down VMware.

Challenge 2: Being one of my first projects in C, memory management presented a real challenge. This presented the bulk of my issues for the project, so I've divided the problems into three primary groups. First, there were a lot of memory leaks. Second, once those were addressed (or thought to be) segmentation faults prevented the program from running correctly. Third, the column validation experienced unexpected behavior.

> **Solution 2:** To address each of these three circumstances, I performed the following:
>
> 1. The memory leaks were caused by this code section:
>   `parameters *data = malloc(sizeof(parameters));`
>   To address this, `free(param);` freed the struct passed by the pointer.
> 2. However, placement of the above solution (located right after allocating the threads in each element's validation function) caused segmentation faults. Therefore, `free(param);` was moved to the end of each element's validation function
> 3. The incorrect column validation was triggered by not placing an additional `free(param);` on the outside of the loop. This final implementation fixed the memory management of element validation functions.
>
> During this process, a debugging `printf()` call was used to determine why the grid was marked as valid despite returning invalid columns. This implementation helped me determine that memory leaks were still occurring, as the `results[]` array returned erroneous values (such as `9`).

```
printf("[DEBUG:RESULT_BITS] - ");
for (int i = 0; i < 27; i++) {
    printf("%d", results[i]);
}
printf("\n");
```

Output [15 points]

Instructions: Provide screen shots of the output produced by your solution with a brief description of each screenshot.

VALID GRID	INVALID GRID
• First it prints the grid being evaluated. • Then each element indicates its validity; first rows, then columns, finally subgrids. • A validation statement of the grid is made. • An optional debug shows the result bits for each `result` in `results[]`	• Same structure as 'VALID GRID' • **EXCEPT**: Replaced [Row 3, Col 2, Sub 3]'s 1 with 9. • Invalid elements are indicated. • The puzzle is indicated as invalid. • Debug results indicate location of invalid elements in the `results[]` array. *Note: Red circle only indicates the position of the invalid integer. It is not part of the screenshot or terminal output.*

```
┌──(kali㉿kali)-[~/CS7349/valid-sudoku]
└─$ ./sudoku
Sudoku Grid:
5 3 4 | 6 7 8 | 9 1 2
6 7 2 | 1 9 5 | 3 4 8
1 9 8 | 3 4 2 | 5 6 7
------+-------+------
8 5 9 | 7 6 1 | 4 2 3
4 2 6 | 8 5 3 | 7 9 1
7 1 3 | 9 2 4 | 8 5 6
------+-------+------
9 6 1 | 5 3 7 | 2 8 4
2 8 7 | 4 1 9 | 6 3 5
3 4 5 | 2 8 6 | 1 7 9

Row 0 is valid.
Row 1 is valid.
Row 2 is valid.
Row 3 is valid.
Row 4 is valid.
Row 5 is valid.
Row 6 is valid.
Row 7 is valid.
Row 8 is valid.
Column 0 is valid.
Column 1 is valid.
Column 2 is valid.
Column 3 is valid.
Column 4 is valid.
Column 5 is valid.
Column 6 is valid.
Column 7 is valid.
Column 8 is valid.
Subgrid 0 is valid.
Subgrid 1 is valid.
Subgrid 2 is valid.
Subgrid 3 is valid.
Subgrid 4 is valid.
Subgrid 5 is valid.
Subgrid 6 is valid.
Subgrid 7 is valid.
Subgrid 8 is valid.

[RESULT:VALID] - Sudoku puzzle is valid!
[DEBUG:RESULT BITS] - 111111111111111111111111111
```

```
┌──(kali㉿kali)-[~/CS7349/valid-sudoku]
└─$ ./sudoku
Sudoku Grid:
5 3 4 | 6 7 8 | 9 1 2
6 7 2 | 1 9 5 | 3 4 8
1 9 8 | 3 4 2 | 5 6 7
------+-------+------
8 5 1 | 7 6 1 | 4 2 3
4 2 6 | 8 5 3 | 7 9 1
7 1 3 | 9 2 4 | 8 5 6
------+-------+------
9 6 1 | 5 3 7 | 2 8 4
2 8 7 | 4 1 9 | 6 3 5
3 4 5 | 2 8 6 | 1 7 9

Row 0 is valid.
Row 1 is valid.
Row 2 is valid.
Row 3 is invalid.
Row 4 is valid.
Row 5 is valid.
Row 6 is valid.
Row 7 is valid.
Row 8 is valid.
Column 0 is valid.
Column 1 is valid.
Column 2 is invalid.
Column 3 is valid.
Column 4 is valid.
Column 5 is valid.
Column 6 is valid.
Column 7 is valid.
Column 8 is valid.
Subgrid 0 is valid.
Subgrid 1 is valid.
Subgrid 2 is valid.
Subgrid 3 is invalid.
Subgrid 4 is valid.
Subgrid 5 is valid.
Subgrid 6 is valid.
Subgrid 7 is valid.
Subgrid 8 is valid.

[RESULTS:INVALID] - Sudoku puzzle is invalid!
[DEBUG:RESULT_BITS] - 111011111110111111111011111
```