Threads of Sudoku

Due Thursday, April 27 at 8 a.m.

CSE 1325 - Spring 2023 - Homework #12 - 1 - Rev 0

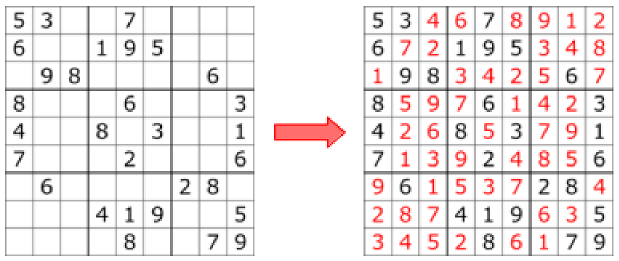
Assignment Overview

IMPORTANT: Homework P12 is provided to you earlier than the expected Tuesday, April 25 start time. Unlike all other assignments, it is due on the *Thursday* before the Tuesday final exam. **P12 will give you critical practice in the use of threads, which is a big part of the final exam. Don't skip it!**

Running multiple threads in a single program is increasingly necessary to take full advantage of the multi-core hardware common in today's market. Let's put those cores to work, and explore how well they perform by solving Sudoku puzzles by brute force!

If you've never solved a Sudoku puzzle, here's the 10¢ tour. The game board consists of 9 x 9 squares that, when solved, contain the digits 1 through 9 in every row and every column. In addition, the game board is divided into nine non-overlapping 3 x 3 squares, each of which must (when solved) contain the digits 1 through 9. At the start, many of the digits have already been filled in. The goal is to fill in the remaining digits to solve the puzzle.

Humans (and I suppose Artificial Intelligence) solve Sudoku puzzles through logic. We'll take the old school brute force approach instead, since our goal is to better understand threads and brute force algorithms tend to work well with threading.



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

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

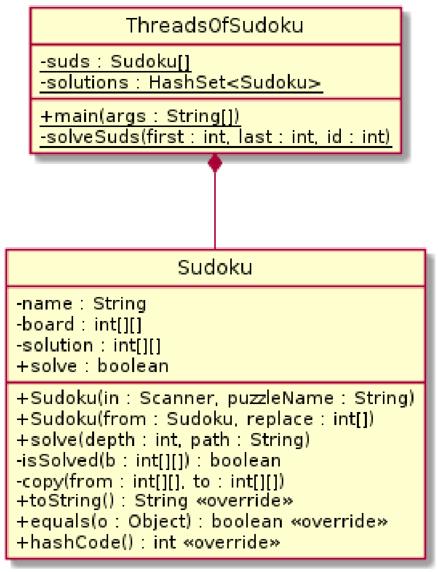
Learn more about Sudoku at https://en.wikipedia.org/wiki/Sudoku. Sudoku images above are from this page and are copyright 2017 by Cburnett, used under the Creative Commons Attribution-Share Alike 3.0 Unported license.

Full Credit

Get the Baseline

I've written the Java code to solve the Sudoku for you - find it at **cse1325-prof/P12/baseline**. Don't get too excited, though - you get to improve my code!

IMPORTANT: **Do NOT modify the Sudoku class at all!** For full credit, you will **modify ONLY the ThreadsOfSudoku class** as discussed below so that it implements threads and then complete a results.txt file that guides you in understanding its performance.



ThreadsOfSudoku
-suds : Sudoku[]
-solutions : HashSet<Sudoku>
+main(args : String[])
-solveSuds(first : int, last : int, id : int)
Sudoku
-name : String
-board : int[][]
-solution : int[][]
+solve : boolean
+Sudoku(in : Scanner, puzzleName : String)
+Sudoku(from : Sudoku, replace : int[])
+solve(depth : int, path : String)
-isSolved(b : int[][]) : boolean
-copy(from : int[][], to : int[][])
+toString() : String «override»
+equals(o : Object) : boolean «override»
+hashCode() : int «override»

Running ThreadsOfSudoku

`usage: java ThreadsOfSudoku threads puzzleFilename puzzleName`

Build the code as usual, then use the Java command line above to run it.

- `threads` - This is an int from 1 to 81 (more on why that's the maximum effective limit in a bit), and specifies how many threads should be created to use in solving the puzzle.
- `puzzleFilename` - ThreadsOfSudoku will open this as a `File` with `Scanner` and pass it to the Sudoku constructor for reading in the puzzle. The file format is the name of the puzzle on a separate line, then 81 whitespace-separated ints representing the puzzle (spaces are 0s). I provided puzzle.txt for you, containing puzzles named 00_zeros to 52_zeros.
- `puzzleName` - Sudoku's constructor will scan through the file until it finds a matching line (indicating the start of the desired puzzle), then read in 81 ints to populate its board[][] array.

The Sudoku Class

Class `Sudoku` represents the Sudoku board as a two-dimensional array of integers - `int[9][9]`. It can also temporarily store a valid solution in field `solution`. It has two constructors - `Sudoku(Scanner in, String puzzleName)` searches `in` until it finds `puzzleName`, then loads the puzzle, and `Sudoku(Sudoku from, int[] replace)` copies `from`, replacing the first zeros it finds from `replace`. The other class members should be self-explanatory, but ask if you have questions.

The ThreadsOfSudoku Class

Class `ThreadsOfSudoku` is the main method you will use to implement threads. Its usage is

`java ThreadsOfSudoku threads puzzleFilename puzzleName` where

- `threads` is the number of threads to create (this is stored in variable `numThreads` but otherwise ignored in the baseline)
- `puzzleFilename` is the file from which to read the puzzle
- `puzzleName` is the name of the puzzle in `puzzleFilename`. All lines of `puzzleFilename` are ignored until a line exactly matching `puzzleName` is found, then 81 integers are loaded into the puzzle file.

The puzzle that is read is then split into 81 (9 x 9) sub-puzzles, by replacing the first and second 0 found with integers in the range 1 to 9, respectively. These new puzzles are stored in static field `Sudoku[81] suds`. It is the 81 sub-puzzles that you will pass to the threads you create - which is also why more than 81 threads isn't helpful!

To help with this, static method `solveSuds` accepts 3 parameters:

- `first` is the index in `suds` to be solved first,
- `last` is the index in `suds` to be solved last, and of course all of the puzzles in between `first` and `last` as well.
- `id` is the identifier for the thread that is running this solveSuds.

`solveSuds` would make a nice thread body, don't you think?

Calibrating ThreadsOfSudoku for Your Machine

If at all possible, take your timings on a machine with *at least* 2 cores - 4 or more is much better (though we won't deduct points if you can't find one). CSE-VM by default has ONE core, although you can configure it to have more. Or you may take your timings on your host OS, Omega, or any other multi-core laptop, desktop, or server you can access.

For this assignment, **you need the baseline ThreadsOfSudoku to run for 30 or more seconds on YOUR machine**. You can adjust the runtime by adjusting the number of "zeros" in the puzzle name. (13_zeros is a puzzle that contains, well, 13 zeros, and it works well on my laptop. My desktop works best with 14_zeros.) List the puzzle name you used in question 1 in your results.txt file. **Use that puzzle for the rest of this assignment - do NOT change puzzles!**

Creating the Threads

Currently, `ThreadsOfSudoku` includes a single call: `solveSuds(0, suds.length-1, 1);` This uses the thread that is running main() to solve all 81 sub-puzzles in `suds`. You will need to:

- Split the sub-puzzles into `numThreads` groups. For example, if `numThreads` is 3, you might split `suds` indices into the 0-26, 27-53, and 54-80. If they don't divide equally, pick a strategy, but **it is important that every suds sub-puzzle be solved exactly once!**
- Create an array or Collection (ArrayList, HashMap, ...) instance to hold your threads.
- Create `numThreads` threads, each running `solveSuds` with one of the subsets of `suds`. IMPORTANT: You *cannot* pass a non-final value into a thread, because that value may change before the thread actually starts. So you will need to create final variables to hold your starting index, ending index, and thread ID to pass to `solveSuds`. For example, `for(int i=0; i<10; ++i) {final int j=i; new Thread(() -> foo(j);}` Because `i` is changing, assign its value to `final int j` and pass `j` into the thread instead!
- Start each thread.
- Join each one back to the main thread.

Protecting solutions

The existing `solveSuds` code stores any solutions found in `HashSet<Sudoku> solutions`. **HashSet is NOT thread-safe!** Make whatever changes are necessary to protect it based on Lecture 24.

That's it! The suggested solution for full credit adds roughly 13 lines of code to `main`, one field, and changes one line of code in `solveSuds` compared to the baseline solution.

Completing results.txt

Now, answer the questions in the **Full Credit** portion of the results.txt file provided in the baseline. You may use the bash `time` command to time your program with different numbers of threads. `time` provides 3 different measures:

- **real** - how much time the user experienced. If you used a stopwatch, you'd measure this time. **This is the time referenced in each question in results.txt unless specified otherwise.**
- **user** - how much time the processor experienced **-** that is, the sum of the time each microprocessor *core* spent working on this program. On a multi-core computer, this may be much higher than the `real` time.
- **sys -** how much time the operating system consumed as overhead. We usually prefer that this be smaller, but a threaded application may require more operating system support than a simple one-thread application. We'll buy big time savings with a little overhead anytime!

What to Submit to GitHub

Add, commit, and push all files.

Include your results.txt file at **cse1325/P12**. Include your source code, my puzzle.txt, and a build.xml file at **cse1325/P12/full_credit**.

Remember this is due on **Thursday -** our only homework of the semester not due on a Tuesday.

Bonus

Let's modify your full credit solution to use a thread pool instead!

Copy your full_credit solution to the bonus directory.

The ThreadsOfSudoku Class

Now, **modify ThreadsOfSudoku to use a thread pool** rather than explicitly dividing the 81 sub-puzzles among threads. Each thread will solve the next available sub-puzzle until all have been solved. Then take the same measurements as in Full Credit, and complete the **Bonus** section of results.txt, comparing its performance and maintainability to your Full Credit solution.

You may approach this in any way you like, but here's a summary of the changes in the suggested solution.

- Create a new static field called `nextIndex`, initialized in-line to 0.
- Create a new static method `getSudIndex()` that returns and increments `nextIndex`. However, after `nextIndex` has reached 80 (the index of the last sub-puzzle), start returning -1 instead. This method will be used by all of the threads in the pool, so **ensure it is protected against thread interference!**
- Modify static method `solveSuds` to accept only one parameter, the thread ID. Instead of looping through a fixed range, call `getSudIndex()` in a while loop and solve that sub-puzzle. Continue solving each puzzle indicated by `getSudIndex()` until the returned index is less than zero.

That's it! The suggested solution deletes approximately 5 lines, adds 4 lines, and changes 4 lines from the full credit solution to produce the bonus solution.

Completing results.txt

Now, answer the questions in the **Bonus** portion of the results.txt file provided in the baseline using the `time` command as in Full Credit. Be sure to add, commit, and push all files in the bonus subdirectory as well as your updated results.txt file in the P12 directory.

Extreme Bonus

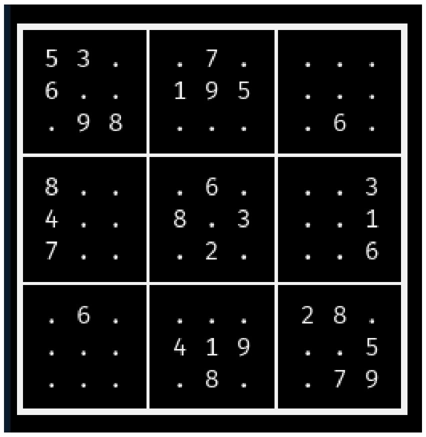
Arrays of bare integers are just hard to read. Since the dawn of mainframes, terminals have been capable of providing very nice looking forms - at first with proprietary character set extensions, but now fully baked into Unicode. Let's explore that capability for our extreme bonus!

The ThreadsOfSudoku Class

Modify Sudoku.toString() to print a fancy puzzle as shown below.

- Replace each zero with a period.
- Use Unicode's Box Drawing characters (0x2500 to 0x25FF) to provide a thick border with thin separator lines for the 3x3 sub-squares.

You should be able to find the Box Drawing character set online with the help of your favorite search engine. If not, you can always ask. We love... well, you know!



```
┏━━━━━━━┯━━━━━━━┯━━━━━━━┓
┃ 5 3 . │ . 7 . │ . . . ┃
┃ 6 . . │ 1 9 5 │ . . . ┃
┃ . 9 8 │ . . . │ . 6 . ┃
┠───────┼───────┼───────┨
┃ 8 . . │ . 6 . │ . . 3 ┃
┃ 4 . . │ 8 . 3 │ . . 1 ┃
┃ 7 . . │ . 2 . │ . . 6 ┃
┠───────┼───────┼───────┨
┃ . 6 . │ . . . │ 2 8 . ┃
┃ . . . │ 4 1 9 │ . . 5 ┃
┃ . . . │ . 8 . │ . 7 9 ┃
┗━━━━━━━┷━━━━━━━┷━━━━━━━┛
```

Your ThreadsOfSudoku puzzle solver should work unmodified with your enhanced Sudoku class.

Be sure to add, commit, and push all files to the extreme_bonus subdirectory.