Concurrent Programming

COMP 409, Winter 2025

Assignment 1

Due date: Tuesday, February 4, 2025
Midnight (23:59:59)

General Requirements

These instructions require you use Java. All code should be well-commented, in a professional style, with appropriate variables names, indenting, etc. Your code must be clear and readable. **Marks will be very generously deducted for bad style or lack of clarity.**

There must be no data races. This means all shared variable access must be properly protected by synchronization: any memory location that is written by one thread and read or written by another should only be accessed within a synchronized block (protected by the same lock), or marked as volatile. At the same time, avoid unnecessary use of synchronization or use of volatile. Unless otherwise specified, your programs should aim to be efficient, and exhibit high parallelism, maximizing the ability of threads to execute concurrently. Please stick closely to the described input and output formats.

Your assignment submission **must** include a separate text file, `declaration.txt` stating "This assignment solution represents my own efforts, and was designed and written entirely by me". Assignments without this declaration, or for which this assertion is found to be untrue are not accepted. In the latter case it will also be referred to the academic integrity office.

Questions

1. A *snowman* is drawn using 3 adjacent and non-overlapping circles of diminishing radius, with colinear centers. In this question you need to fill an image with random, non-overlapping snowpeople of varying sizes, using multiple threads. **10**

   Note that it is up to you how to draw circles. You can explore the Java APIs, or draw it pixel-by-pixel with an algorithm of your choice (e.g., the mid-point circle algorithm or Bresenham's algorithm are straightforward and fast).

   Each snowman figure is defined by two values: a single size value, and an orientation. The size value represents the radius of the base (largest) circle, with the two progressively smaller circles defined by some fixed reduction in radius (your choice). Orientation is chosen randomly, as one of four possible directions to stack the circles—up, down, left, or right.

   Each thread repeatedly attempts to draw a fixed number of snowpeople. For each snowman, it chooses a random position in the image, size value (integer ≥ 8), and orientation, and ensures the resulting snowman at that location will fit within the image bounds and will not overlap the circle perimeter of other drawn (or in-progress) snowpeople. Importantly, once a thread starts drawing the snowman it must fully complete the process—no erasing of previously drawn pixels! You can use any colour (visible) to draw your circles. Once all threads complete their work the image should be output, as a file named `outputimage.png`.

   Provide a program, `q1.java` that accepts the following command-line arguments, w, h, t, and n. It should launch t threads to each draw n/t snowpeople on a $w \times h$ image. You can assume n is divisible by t.

   Template code including basic image creation, pixel-setting, and saving is provided in MyCourses. Add timing code (using `System.currentTimeMillis`) to time the actual work of the program (including all threads, but excluding all I/O). The time taken in milliseconds should be emitted as console output. No other console output should be included.

Now, for some reasonable choice of n, w, and h, plot total time versus t. Verify that $t = 1$ takes measurable time (at last a few milliseconds). Then try increasing values of t. Note that when timing the program you should execute the exact same execution scenario several times (at least 5), discarding the first timing (as cache warmup), and averaging the rest of the values.

Provide a graph of the relative speedup of your multithreaded version over the single-threaded version for the different choices of t. *Your code must demonstrate speedup for some value of* $t > 1$, but perhaps not all values. This of course does require you do experiments on a multi-core machine.

Either as text included in the plot file, or as a separate `.txt` file (and specifically *not* just as code comments), briefly explain your results in relation to your synchronization strategy and the number of cores available on your test machine. Note that achieving speedup is not necessarily trivial: beware of sequential bottlenecks from different methods and APIs you may use or implement, and choose an appropriate strategy.

Provide your source code (`q1.java`), and as a *separate document* your performance plot with a brief textual explanation of your results (why do you get the speedup curve you get).

2. The classic board game "Snakes and Ladders" (or "Chutes and Ladders") is based on a 10x10 grid. A player starts in the lower-left grid cell, and based on a random roll (in the range 1–6) advances that many grid cells horizontally, potentially including moving to the cell in the next row upward if they would go off the board horizontally. Horizontal movement changes direction every row, following a boustrephedon (zig-zag) pattern. Once they reach (or go past) the top-left cell they win. Challenge is added by the presence of some number of "snakes" and "ladders", each of which defines a connection between two cells on different rows (and perhaps different columns). A player that lands on a cell containing the tail of a "snake" moves downward to the cell indicated by the head of the snake. Conversely, a ladder moves a player upward—landing on a cell containing the base of the ladder moves the player to the cell containing the top of the ladder. **10**

   In this task you will simulate concurrent game play and board modification. First, build a board consisting of a 10×10 array of CELL objects. Initialize the board with 10 snakes and 9 ladders, randomly created, with the condition that snake heads and tails and ladder tops and bottoms must always refer to different cells and never to the starting or ending cells, and snakes (and ladders) must always go between different rows.

   Once the initial board is created define and start 3 threads as follows. Each thread should keep a log of the operations they perform, and the timestamps (in milliseconds) associated with them. Pre-populate the adder log with the initial snake/ladder additions.

   - One thread plays the game: the *player* thread starts at the beginning, chooses a random number in the range 1–6, and advances that many cells as described above. If it arrives a cell containing a snake tail it moves to the corresponding head cell, or if it arrives at cell containing a ladder bottom it moves to the ladder top. After each move it sleeps for 20–50 ms (duration randomly chosen). Once it reaches or would go past the top-left cell it sleeps for 100 ms, and starts again.
   - One thread is in an infinite loop adding snakes or ladders. The *adder* thread adds a snake or ladder (equal probability) between 2 random cells on different rows, verifying they are valid end-points (not used by other snakes or ladders and not the starting or ending cell). Once it succeeds it sleeps for k ms.
   - One thread is in an infinite loop removing snakes or ladders. The *remover* looks for either a snake or ladder and removes it from the board. Once it succeeds it sleeps for j ms.

Your code should accept parameters k, j, and s, where s is the number of seconds to let the simulation run. After s seconds have passed all threads should stop execution. Sort the thread logs by timestamp, and emit an ordered list of operations, as per the following (note that cells are uniquely identified by their $y * 10 + x$ coordinate, 0-based of course, and the comments are just here to describe the format).

```
000000000 Adder snake 12 45    // initialization of snakes and ladders
000023204 Player 23            // player moved from previous cell to cell 23
000023204 Player 23 47         // player snaked/laddered from 23 to 47
000023405 Player wins          // player won
000023444 Adder snake 34 67    // added a snake between 34 and 67
000023445 Remover ladder 11 44 // removed a ladder between 11 and 44
```

Verify your program runs correctly for different values of k, j, running each for a few seconds. Provide sample output as well as your code.

What to hand in

Submit your declaration and assignment files to *MyCourses*. Note that clock accuracy varies, and late assignments will not be accepted without a special permission: **do not wait until the last minute**. Assignments must be submitted on the due date **before midnight**.

Where possible hand in only **source code** files containing code you write. Do not submit compiled binaries or .class files, but do include a `readme.txt` of how to execute your program if it is not trivial and obvious. For any written answer questions, submit either an ASCII text document or a .pdf file. Avoid .doc or .docx files. Images (plots or scans) are acceptable in all common graphic file formats.

This assignment is worth 10% of your final grade. **$\overline{20}$**

Programming assessment

	Mastery	Proficient	Developing	Beginning
Correctness	The solution works correctly on all inputs and meets all specifications.	The solution meets most of the specifications; minor errors exist.	The solution is incorrect in many instances.	The solution does not run or is mostly incorrect.
Readability	Well organized according to course expectations and easy to follow without additional context.	Mostly organized according to course expectations and easy to follow for someone with context.	Readable only by someone who knows what it is supposed to be doing.	Poorly organized and very difficult to read.
Algorithm Design	The choice of algorithms, data structures, or implementation techniques is very appropriate to the problem.	The choice of algorithms, data structures, or implementation techniques is mostly appropriate to the problem.	The choice of algorithms, data structures, or implementation techniques is mostly inappropriate to the problem.	Fails to present a coherent algorithm or solution.
Documentation	The solution is well documented according to course expectations.	The solution is well documented according to course expectations.	The solution documentation lacks relevancy or disagrees with course expectations.	The solution lacks documentation.
Performance	The solution meets all performance expectations	The solution meets most performance expectations	The solution meets few performance expectations	The solution is not performant.