**TINASHE MUKUNDI CHITAMBA**

**CHTTIN007**

**ASSIGNMENT 2**

**INTRODUCTION**

In order to make full use of a computer’s resources when running a resource heavy program, it is sometimes necessary to use multithreading, such that multiple threads operate on a part of the program concurrently. However, if not implemented properly, multithreading can come with its unique set of problems and bugs, which might either give undesired results, or make the program crash altogether. It is therefore important as a programmer to make sure that your program is thread safe, by making sure that access to shared resources is managed properly. This report documents how I was able to make sure that the Flow simulation is thread safe, as well as describing how I got to make the program run.

**CLASSES USED**

Flow

I added three buttons to the existing Flow class, a Play button to start the simulation, a pause button to pause the simulation when its running, and a reset button to clear the water on the terrain and reset the timer. I added button listeners to these buttons to trigger the desired responses. I also added a counter, which shows the current timestamp of the simulation. All threads are synchronized such that one increment of the timestamp means that all four threads have completed one step of the simulation.

TerrainWater

The terrainWater class represents the water in the simulation. It has a bufferedImage as an instance variable, which starts off as transparent (no water on the terrain) and each grid position turns blue when water is added to it. The class also has two arrays of type float, one for storing the depth of water at a grid point, and one storing the surface level at a grid point. The surface level is the height of the terrain at that grid point plus the depth of water at that point. The class also has specific methods that are the main drivers of the simulation.

* The isMinima() method checks if a grid point is a local minima, that is, the lowest point relative to surrounding grids.
* The flow() method takes in the coordinates of a grid point, and if that grid point contains water, determines which grid point it should transfer water to.
* The waterFlowTo() method takes in coordinates for two grid points, and transfers water from one grid point to another. The method also changes the colour of the grids. This method also uses the synchronized keyword, so that only one thread can access it at any point in time, to prevent data races.
* The addWater() method is used when water is added by the user on click. It updates the values in the respective arrays, and changes the colour of the grids to blue.

flowThread

The flowThread class extends java.lang.Thread, and is the one that is used to run the simulation using multiple threads. It takes in a start and stop integer, a terrainWater object, and a Terrain object. The run() method overridden uses the permuted array from the Terrain object, and loops through the grid points between the start and stop integers, calling the flow() method on each of the coordinates.

surfaceLevel

The surfaceLevel class contains a depth value representing water, a height value for the height of a terrain, and a surfaceLevel value representing the surface level at a point.

**THREAD SAFETY and CONCURRENCY FEATURES**

Concurrency is correctly and efficiently managing shared resources accessed by multiple threads, sometime simultaneously to avoid data races and deadlocks. To ensure that my program was Thread safe, I made use of synchronization and atomicBooleans. In the waterFlowTo() method of the terrainWater class, I used the synchronized keyword to make sure that only that one thread can call the method and make changes to the shared resources (the depth array for water level and the surfaceLevel array ). This makes sure that two threads do not write the same value to a grid point twice, where it was supposed to write once, thereby “creating” water. Water must be conserved during the simulation, so the amount of water added before the simulation starts should be equal to the amount of water cleared at the edges plus the amount of water left in local minima. The synchronized keyword makes use of locks, hence when the simulation is running, one thread attains the lock of the method flow(), and any other thread will have to wait for it to finish. The thread then writes onto the depth and surfaceLevel arrays and gives up the lock after it is done.

I also made use of the atomicBoolean, that is, to stop and start the while loops that carry out the simulation. This ensures that there is consistency in starting and stopping the simulation between the different threads, as the most recent value of the Boolean is returned every time it is accessed.

To make sure that my swing components were thread safe, I made sure that the logic of my simulation is not done in the view class. This way, the view is handled by one thread, and the logic of the program is handled by other threads, and only updates the view, that is, the GUI. This is part of the implementation of the model view controller software design pattern that I will describe below.

**LIVELINESS**

To ensure that my program is lively and efficient, I chose not to add locks to the isMinima() method as well as the flow() method. Doing this would make the program close to sequential, and the gains of concurrency and parallelism will be lost. The absence of locks in these methods however, did not cause data races, as the two methods do not involve writing to any variable. The way the simulation works is in such a way that even if a Thread A changes a grid position while grid position B is reading the same position, the updated version of that grid point will be catered for in the next iteration of the thread.

**THREAD SYNCHRONIZATION**

For my simulation to be consistent, I had to make sure that all my threads are synchronized at every timestamp. This means that all threads should begin the next step of iterating through the grid points once they are all done with the previous iteration. To do this, I had a while loop in the run method of my FlowPanel class which at every iteration would create four threads (flowThread) that would iterate through different parts of the permuted list, and use the join() method to synchronize them so that they are all done and synchronized at every step. The counter at the right side of the GUI shows the number of steps the simulation has taken.

**NO DEADLOCKS**

A deadlock is a situation where one thread A holds a certain lock, and waits for thread B to release another lock, but thread B waits on the lock that thread A has. My simulation is safe from deadlocks as there is only one method which uses the synchronized keyword, so no two threads will have 2 different locks at the same time.

**VALIDATION OF SYSTEM**

To validate that indeed my simulation was working, and that it was free of data races and deadlocks, I added counters in the testing phase , to keep a record of the water that has been added by the user, the water that has been cleared from the terrain, and the water that is left on the terrain. If the water added before the simulation started was equal to the water cleared plus the water left, then water was conserved. This means that no additional water has been created or lost due to data races. Below are the results of the tests.

**MODEL VIEW CONTROLLER**

The MVC design pattern was followed in creating the simulation. In the MVC pattern, the **Model** is the data of the program, and it updates the view. In my program, the Terrain class, terrainWater class, and surfaceLevel formed the model. The **view** is what the user sees and what the user interacts with, and in this case, it was the Flow class, which sets up the GUI and creates the required buttons and swing elements. The **Controller** is the link between the model and the view, as it manipultes the data in the model when prompted by the view. In my program, the flowPanel class and the flowThread class form the controller, as they manipulate the model when prompted by the view.