Assignment 2

Introduction

This aim of this assignment was to implement a multi-threaded water flow simulation that shows how water on a terrain flows downhill, accumulating in basins and flowing off the edge of the terrain.

Code Description

I added two more classes in order to make the necessary simulator, GridTings and TempleRun. I have also added a few things in my Flow class.

GridTings

This class is meant to take in all the grid positions of the water as it flows into the basins and eventually disappearing. It is also the one that controls the amount of water that flows and where it flows.

This class has a constructor that takes in 3 parameters: `RowIx: row number, ColIx: column number and height: height value. It also sets the H2OUnits to zero.`

I am going to discuss the methods in the class and explain what they do and why they are the way they are.

getRowIx, getColIx and getHeight: We need methods for us to know the positions of grid and height at any point given and time and these methods do exactly that. These methods are meant to get the row and column indices and height respectively whenever they are called.

I did not use any thread safety features on these methods because these are immutable values. Immutable values do not need any thread safety features because no thread is going to change them, so they are safe for simultaneous use by various threads because the data each of these threads use is not mutable and hence won't affect the results.

getH2OSurface and getH2OUnits: The get water surface method does exactly what its name says, gets the water surface. It is calculated as the sum of the specific height in question and the result of multiplying the specific H2OUnits in question and 0.01.

The getH20Units method is used to get the water units(H2OUnits) at a specific grid point at the time when it is called.

Both these methods were synchronized because the values they hold are mutable and they can be modified by different threads at the same time leading to inaccurate values

addH20 and removeH20: These methods add and remove specific number of water units (numUnits

****(clarification of num and h20 units)

respectively when called upon

These methods are synchronized because the adding and removal of water units(numUnits) must be sequential in order for us to not get any race conditions and Heinzen bugs.

resetH20: This method resets the H20units to zero. Basically, it restarts the program (GUI). And when it restarts it kills off all existing threads and resets everything to 0.

It was synchronized because we need to do it after all the threads have finished or basically as a complete reset of the program to avoid scenarios where a thread for a different execution cycle continues to execute in a different cycle. This synchronization of the methods ensures that all threads of the same cycle are executing in the right cycle.

toString: Every class has to have a toString so I added this one.

Now I am going to discuss the other class which is called TempleRun

TempleRun

This is the code that controls all the threads individually. It is the one with all the details on how each individual code is supposed to run.

The class has a constructor that takes in 4 parameters: `lowlife: the lower index that is visible to the thread, highlife: the upper indx visible to the thread, LD: terrain to analyse and indx: index the thread has access to in the finishedStep array of FP. The constructor also sets the Boolean isRunning = true; and int cnt to zero (it keeps track of the timing.`

I am going to discuss the methods I have used in this class.

Run: This method only runs when the Boolean isRunning=true. It first checks if the cnt value has been incremented. The method takes in the portion of the grid the program must work as it will iterate over it and selects the grid indices that have been selected by the user.

Generally, this method controls the flow of the water through the terrain and where it goes. It leads it to the areas of lower height as it should be in real life.

Synchronization locks are put on certain areas of the code like (current item) and LN as a way of trying to avoid race conditions and deadlock.

Flow

I have added buttons in this class as they were missing. I have used and synchronization locks and Atomic variables in order to ensure thread safety within my code. The areas that I have added the Atomic variables and locks are areas that I feel needed them for me to cut down on race conditions.

Java Concurrency Features

3. Code Structure to ensure Concurrency

3.1. and 3.2 Thread safety and Synchronization

I ensured thread safety by making sure that no two threads access the same shared memory at the same time using our synchronized method. This only allows one thread to access the memory alone during each execution time and soon as it is done another thread follows. I did not use any nested synchronized blocks in order to eliminate risk for deadlock as no circular dependencies would arise. I used atomic variables and synchronization in order to protect the data that is accessed by multiple threads and has potential to cause race conditions for my code. Leaving these data items unprotected would have resulted in inefficient function of the overall code and incorrect outputs on the GUI. The reason for this is that threads access the same shared memory, and multiple threads

may access the shared memory simultaneously. If this is the case (especially with my code) certain threads may pick up updated values of a variable, when in fact they needed to original value. Synchronization helps to prevent this since it bars two threads from accessing the same object lock at the same time. Moreover, volatile variables are accessed through main memory, rather than CPU caches (hich takes more time), which allows each thread to work with the same data (if thread 1 adds 5 to a total in its CPU register, thread 2 will not know this because the value will be unchanged in its CPU register). Thus, the combination of synchronization and volatile is best, which is what occurs with Atomic variables.

3.3. Liveness

Liveness is when a concurrent program executes and finishes in a timely manner. My program makes use of synchronization and atomic variables to prevent any deadlock or lengthy periods of execution. This allowed my program to run as intended, and finish when the reset button was pressed. It finishes in a timely manner.

3.4. No Deadlock

When two or more threads are blocked indefinitely, that's called a deadlock. This happens because these two threads both require data from each other to continue. This mainly arises when your architecture uses circular dependencies, or when a synchronous method calls another synchronous method. I designed my code to prevent deadlocks from occurring since there is no circular dependences in my code, nor do any threads ever require information from another thread to continue.

4. System Validity

Proving a lack race conditions or other concurrency related issues in a multithreaded program a near impossible task. Running the program multiple times does not necessarily prove anything and using software to test if the program changes the way it executes does not give an accurate dissection of the matter. Moreover, any bugs found could disappear completely when the program is run again. However, testing was attempted by running the program multiple times with two datasets and observing the behaviour. The threads were also run at different speeds. Changing the speed meant changing the time for which the thread was slept. Sleeping a thread or using yield () give control to another thread waiting to run. Therefore, extra yields were added in the thread operation to see if any bugs occurred. It should be emphasised that although no concurrency related bugs were found, this does not mean that the program is bug free. It could just take the right (or wrong) order of threads to break it.

5. Model View Controller

The Model-View-Controller pattern separates the code operations into three well-defined categories. The View is the GUI of the application. It takes the user's input and sends it to the Controller. The Controller takes that input which in turn distributes it to the Model system which performs the operations. It should perform all required operations and then sends the result back to the controller. The controller then sends it to the view which finalises the output.

In the case of this assignment, the View are the Flow and FlowPanel classes. The Model comprises of the Terrain and GridTings classes. The Controller class is the TempleRun class.

Conclusion

I therefore conclude that I have made my concurrent code and it is running efficiently enough for you to say that it runs well.