CSC2002S ASSIGNMENT 2

CONCURRENCY

MFNLIN003

Introduction

The aim of this assignment is to design a multithread java program that ensures thread safety and concurrency for the application to function well. To archive thread safety synchronization was used which is capable to control the access of multiple threads to any shared resource. In the case of the assignment this was implemented to enable a transfer of water and pausing of the process to happen separately.

Concurrency

Is the ability to run several programs or several parts of a program in parallel. If a time-consuming task can be performed asynchronously or in parallel, this improves the throughput and the interactivity of the program

Multithreading

Is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process. Threads are lightweight sub-processes; they share the common memory space. In Multithreaded environment, programs that are benefited from multithreading, utilize the maximum CPU time so that the idle time can be kept to minimum. Threads can be created by using two mechanisms

1. Extending the Thread class
2. Implementing the Runnable Interface (this is the way implemented in the assignment)

Expected Application Behavior

To archive the assignment objective further develop the parallel programming application from assignment 1 to construct a grid terrain using the text files containing details of the terrain. The application is supposed to make using of the Swing class to contact a Graphical User Interface (GUIs). The GUI is supposed to display the grid terrain landscape image as a greyscale image. By clicking on the window water should be added to the terrain as an overlay of the terrain image. By clicking the play button, the simulation of water is supposed to be evoked. Water is transferred to the lowest point among the surrounding points when the pay button is pressed. When the pause button is pressed the simulation is supposed to pause. When the reset button is pressed all water is to be cleared from terrain and the end button disposes the window. There should be a Time step counter displayed on the window counting how many time steps it takes to complete each simulation.

Classes modifications and Methods

Several modifications were made to the *FlowSketon* package classes in order to archive the objective of the assignment. A new classes Water class and Producer class were created. Below are the detailed explanations of the modification. A link to the *javadocs* for the Classes and methods for assignment is here: **doc**

1. Terrain Class:

   The only modification added to the Terrain class is the *getHeight()* accessor method. The method is used to retrieve the terrain elevation at a given point based on the coordinates passed as parameters.

2. Water Class:
   The Water class is used to add water to the grid terrain by creating a buffered image with blue colored pixels and overlaying it on the greyscale terrain image. It is also used to transfer water to the lowest surrounding point.

   The class has instance variables *t* (Terrain object), *depth* (depth of water added to each point), *water* (Buffered image of water to be added to grid terrain) and *blocks* (ArrayList that stores an array is size two contains the coordinates of the points added water to). The Water class constructor has the same name as the class name. Its purpose is to initialize the object of the Water class by executing the java code and the buffered image by setting it to the same size the terrain image. The blocks array is initialized.

   addWater() method it is a setter method since it does not have a return type. It takes the coordinates of point as parameters these coordinates are added to the *blocks* ArrayList. First check if the use state is true then if it is the change sin the water image must be implemented else they should not. check if water has not yet been added to that point already by making use of the *BufferImage.getRGB(x,y)* method that returns the integer value of the color at that pixel point. If the value does not meet the blue color int value, then water must be added in the 3 by 3 area of the point by coloring every pixel on water image blue using a loop.

   convertWater() method it is setter method since it does not have a return type. It is used to convert the unit of water to its double value as one unit represents 0.01m of water.

   getWaterSurface() method is accessor method since it returns a float value. The method is used to return the water surface which is the grid elevation plus the water depth at that point.

   transferWater(ArrayList) method is a setter method since it does not have a return type. It takes in the ArrayList as a parameter which contains potential points where water is going to be transferred. The use state is checked if its false then the changes in the window should show the water transfer simulation else it should not. The method is used to iterate through ArrayList and use the *getPerumate()* to get 2D coordinates of a point and store it in the coordinates array. If the coordinate array is contained in the blocks ArrayList it means, there is water on that point. If so, water is supposed to transferred to the lowest point surrounding the current point by coloring the current point transparent, removing one unit of water from it, coloring the lowest point to blue and coloring the lowest point pixel blue. This is archived using of the *BufferImage.setRGB()* method. However, water should not be transferred to the edges of the grid point, so each point is checked if it is not at the edge.

   getImg() method is an accessor method since it returns the water image.

removeWater() is a setter method since it does not have a return type. The method is supposed to remove all water from terrain by coloring all pixels in the water image to translucent. This is achieved using *Color.TRANSLUCENT* and *BufferImage.setRGB()* methods.

3. Producer Class

   The class implements the *Runnable* interface. The purpose of this class is to split the permute array into 4 and depending on which thread is running at that point that determines which quarter of the ArrayList will be used to transfer water to random lowest surrounding point in the grid.

   It has four instance variables Water object w, Thread object thread, and the thread number. The instance variables are initialized in the *Producer* constructor which has the same name as the class. *Terrain.permute* ArrayList is populated with linear index positions to allow random traversal over terrain during water transfer (*w.t.genPermute()*).

   run() method is an overridden method from the Runnable interface. 4 for quarters of the *Terrai.permute* arraylist are created named *first, second, third ad fourth*. The arraylists are populated with is populated with linear index positions. If the *threadnum* is equal to one, then the *first* arraylist is going to be used by the first thread to transfer water. If the *threadnum* is equal to two then the *second* arraylist is going to be used by the second thread to transfer water. If the *threadnum* is equal to three then the third arraylist is going to be used by the third thread to transfer water. If the *threadnum* is equal to four then the *fourth* arraylist is going to be used by the fourth thread to transfer water.

4. FlowPanel Class:

   A couple number of modifications were made to this class below is a detailed description to the modifications.

   The instance variable in the class is now the Water object (*water*) instead of the Terrain object. This change was made to ensure that the water methods like the *Water.getImg()* overlay the water image over the terrain image before and after the transfer of water. Two AtomicBoolean variables are created named *running and end.* The variables are used to control the water transfer simulation. An array of *Producer* and *Thread* objects are created of size 4.

   Flow panel constructor has the same name as the class used to initialize the Flow Panel object. By intialising the Water object, the AtomicBoolean variable *end* by setting it to true

   paintComponent() method the only modification in this method is by calling the *Water.getImg()* method that returns a bufferedImage of the water image and drawing the image to the graphics component by overlaying it on top of the terrain image passed in the water class .

   getTimeStep() is an accessor method it returns the number of Time step simulations done by the 4 threads.

   run() method is setter method that controls the water transfer simulation. 4 Producer and Thread objects are created and initialized using a for loop. The index of the array determines the thread number and name of each thread. The simulation iterates if the end state is not true. If running variable is set to true, the current thread is locked using synchronized(this) statement. If the end state changes the loop should break out of the current loop. A for loop is used to run the Threads i.e. evoking them to execute. The *Producers.run()* is the one being used to start the threads. This method is the one that evoked the transfer of water and splitting of work between the 4 threads. The results from the threads are joined using the *Threads.join()* method. Therefore, the main thread was made to wait while the other threads before executing the results which is the flow of water across the terrain. When the waiting time is up the thread is unblocked and proceeds from where it left off. The atomicInteger is incremented to show the end of a water transfer simulation. The window must be repainted at it run to show the change in water location.

5. Flow Class:

This is the main class since it houses the main method which is the entry point where the process of execution starts. A few modifications where made to the class mainly in the *setUPGUI()* method.

A Water object named *w* was created which takes in the depth of 3 units and Terrain object *landdata* as instance variables. A JLabel called *timestep* was created. It is supposed to display how many time steps it takes to complete each simulation by the 4 threads. To archive this *FlowPanel.getTimeStep()* method is called that returns the integer value of the time steps.

A mouse listener was added to the added to the Flow Panel *fp* to enable the adding of water blocks in grid and the position clicked by the mouse. The in *Water.use* should be set to true to show water addition on window. The water should not be added to the outer points of the grid. Three more buttons were created *reset, play and pause* and added to the JPanel *b*. All the buttons have an action listener just like the *end* button, but their actions differ unlike the *end* button that is supposed to dispose the frame when clicked.

The reset button is supposed to clear all water blocks from the terrain. This process it done by calling the *Water.removeWater()* method that colors every pixel transparent in the water image and the *FlowPanel.repaint()* that repaints the image component.

The play button is supposed to evoke the start and resume of the water transfer simulation. This is achieved by changing the running state to true (*fp.running.set(true)*) which sets the running state to true in order to start the water transfer simulation. It sets the pause state to false and unlocks the blocked thread to show that the other thread has finished its execution meaning its run returns. The *Water.use* is set to false so that the transfer of water is shown on window.

The pause button is supposed to pause the water transfer simulation. This is achieved by using the *fp.running.set(false)* that sets the pause state to true.

Implementations

Since java does not allow multiple inheritance the more convenient way is to implement the *run()* method from the interface Runnable instead of inheriting from thread this was done by implementing *Runnable* interface in the *FlowPlannel* class and the Producer class.

1. <u>Thread safety</u>
   It was ensured in declaring *Producer.run()* method as synchronized to avoid race conditions. Race condition   is a bug in a program where the output and or result of the process is unexpectedly and critically depended on the relative sequence of other events, we just wait for all threads to complete their processes. This block and unblocks threads in water transfer simulation process. The processes of pausing and running of water are supposed to happen separately for the application to function well to allow threads to coordinate. So, AtomicBoolean variable running controls the state in where the thread should be blocked or not i.e.(water should run or pause) .It is changed when the play button is pressed to true and when pause button are clicked changed to false. If the pause state is on the thread is block if not it is unlocked which causes one thread to wait until another has terminated. The Thread.join() method is used to make sure that there is a wait for all threads to finish executing before displaying the result which is the transfer of water

   To avoid bad interleaving which is when a wrong result is displayed due to unexpected interleaving of statements in two or more threads, mutual exclusion is implemented. It is used to make sure the 4 threads do not try to access the same operation at the same time. This is archived by blocking the current thread by using the synchronized(this) statement in the FlowPanel.run() method. The synchronized context hey should be called in a loop to check the conditions i.e. Whether the play or pause button has been pressed. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. All the other threads then wait until the first thread come out of the synchronized block. No other thread can access the *Producer.run*() method when the other thread is using it.

2. <u>Liveness</u>
   A concurrent application's ability to execute in a timely manner is known as its liveness. The use of the for loops used to run each thread at a time archives this and the use of the *Thread.join()* makes sure that all threads finish before transferring water.

3. <u>Deadlocks</u>
   This is a situation where all threads are blocked. To avoid this, I used only one lock for the water transfer simulation and the lock is always acquired in the same order (*synchronized run()* in the Producer class)

4. <u>The Model-View-Controller pattern</u>
   Is a software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements. This is done to separate internal representations of information from the way’s information is presented to and accepted from the user

   Model
   The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application. In this assignement this must be the FlowPanel.run() method which used the states controlled by the pause and play button to determine which thread should be blocked and unblocked to enable the water transfer simulation to run.

View
Any representation of information such as a chart, diagram or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants. In the assignment code this would probably the Flow.setUPGUI() method and the *FlowPanel.paintComponent()* which is responsible for the setup of adding the components of the GUI to the frame for user viewing.

Controller
Accepts input and converts it to commands for the model or view.In this assignment this would be the Flow.setUPGUI() method since it the one with ActionListeners and MouseListeners when evoked and change the view.

Results

There is a text file named *README* it has further instructions on how to run the assignment applications in case the *makefile* is not functioning well to run the applications because the programs have too many dependencies,

From the applications codes given the water transfer simulation does not seem to be working despite the thread safe multi-threading methods used to make sure concurrency occurs. Several thread safe multithreading methods were implemented altering this code to make sure it carries out its task, but nothing was working. Methods like use of synchronization and locks, atomic variables and reentrant locks were tried none of them were working properly. At some point water would start flowing as water blocks were being added to the terrain (which is not the case in the code submitted but it was a previous occurrence before) or like in the current situation nothing is happening at all. The producer consumer problem for multithreading was also an attempt to try solve this assignment but it does not seem to have been the solution since it applies to two threads mostly one receiving one giving unlike in this assignment all four threads are carrying out the same process. Just not at the same time one must wait for the other to finish then carry on. The other processes (i.e. removing water, ending, adding water) are responding to their purposes except for the pausing and running of water process.

However, it does seem like multithreading implementation is in the right manner the problem seems to have been in the calling of the water image from the water class. The window is just showing the adding of water not the to the terrain in the form of the layered image but the changes implemented by the *tranferWater()* method are not being shown. The use of an atomic variable *use* which was created as a form of control of whether the changes to be shown or not. It was set to true on mouse click to show that the adding of water should be shown and it was set to false when the play button was clicked to implement the water simulation in the in the window and the water image pixels were supposed to be colored continuously. But those changes were not shown in the simulation. This is most probably where the error.

The git log for all commits is stored in a text file named a.txt. In case is it not there below is the git log:

```
commit db95df87c1d06719969f281d2841451c895810cf

Author: Liz-cloud <mafunulinda@gmail.com>

Date:   Mon Sep 21 03:23:41 2020 +0200

    10th

commit 5d7ab9953bd8a20d6707958b6daf8f7261f79bca

Author: Liz-cloud <mafunulinda@gmail.com>

Date:   Mon Sep 21 03:09:40 2020 +0200

    10th

commit db3be0b73b2a7c79e3de3fc2cde16ec34418a753

Author: Liz-cloud <mafunulinda@gmail.com>

Date:   Mon Sep 21 02:51:59 2020 +0200

    9th

commit 6e58a7e19c171b6131d5965c9c1cb7d840999899

Author: Liz-cloud <mafunulinda@gmail.com>

Date:   Mon Sep 21 02:51:38 2020 +0200

    9th

commit e4aaebb7cd76211070f237132ae463abd8f0d283

Author: Liz-cloud <mafunulinda@gmail.com>

Date:   Mon Sep 21 02:36:04 2020 +0200

    8th

commit ce908eba966da2df0c97682a5fbcdd4be9e6d39a

Author: Liz-cloud <mafunulinda@gmail.com>

Date:   Mon Sep 21 01:46:45 2020 +0200

    7th

commit ba1384be3dfbc7659bdd491b813163824f05af9e

Author: Liz-cloud <mafunulinda@gmail.com>

Date:   Sun Sep 20 23:43:26 2020 +0200

    6th

commit b5084114afb58a4f8582278a1694a256227b6f93

Author: Liz-cloud <mafunulinda@gmail.com>

Date:   Sun Sep 20 01:38:20 2020 +0200

    5thh

commit 24510a06762a2b03980373b08af03849352c4596

Author: Liz-cloud <mafunulinda@gmail.com>

Date:   Sun Sep 20 01:38:09 2020 +0200

    5th

commit 7f492fbc52fedfb99e61ff1dddb7ffac421aab6d

Author: Liz-cloud <mafunulinda@gmail.com>

Date:   Sat Sep 19 18:44:20 2020 +0200

    tired

commit b1cd7a39f73018bae13cf04c63983d80532caf75

Author: Liz-cloud <mafunulinda@gmail.com>

Date:   Sat Sep 19 18:44:12 2020 +0200

    tired

commit 99cf7fe5d3df2065788edaf7548f19787ed852d1

Author: Liz-cloud <mafunulinda@gmail.com>
```

Conclusion

Despite the endless readings, research from various sources carried out for the purpose of the assignment, further practice examples are needed on this topic in order to grasp the concept since the application did not achieve its purpose. So, it cannot be conclude to say the process of safe multithreading did work or was it was the problem with the displaying of the image.