UNIVERSITY OF CAPE TOWN

CSC2002S

COMSCI

Concurrency

Author:
Yaseen Hull

Student Number:
HLLYAS001

October 4, 2018

Contents

1 Introduction

Akin to the concept of parallelism, concurrency is about multiple threads accessing shared resources. In concurrent programming we have threads mostly operating in an uncoordinated way. Newly created threads run alongside other threads and their operations become interleaved. Hence multiple threads may access and/or modify data at the same time. Concurrent read/writes can lead to issues such as data races or bad interleaving which may cause the program to act strangely. In some cases these bugs may be very hard to find and could possibly only occur after the one millionth iteration. Therefore it is important to ensure thread safety by controlling access to shared memory using concurrency concepts such as synchronization, volatility and atomicity.

1.1 Problems with concurrency

Concurrent programming faces a problem because of its non-deterministic nature. As mentioned before multiple read and writes to a shared resources can result in bugs known as race conditions. These race conditions can be categorized into bad interleaving's and data races. An interleaving in which multiple threads operate non-sequentially can often be bad. These bad interleaving's are referred to as a "high-level race" and depend on what the programmer is trying to do.
A data race on the other hand is a sort of "simultaneous access error". There are two kinds of data races, the first being when one thread reads an object field while another thread writes to the same field. The second is when two simultaneous writes from two threads to the same object field occurs. Other problems that could occur is the situation of deadlock in which threads are blocked forever, all waiting for another thread to do something.
A programming model requiring concurrent access to shared memory of which the contents are mutable might be troublesome due to race conditions that could occur. However, we can implement the concept of mutual exclusion which is to enforce that only one thread is allowed access to an object field at any given time. To do this synchronization primitives provided by the programming language is used. Essentially this primitive checks if the object field is being operated on by another thread and if not sets the current thread to lock the resource from other threads operating until its terminated or finished its operations. To counter deadlock our concurrent program must take and an approach between fine and course grained models. Fine grain implying that more locks are used on fewer objects and coarse being fewer locks on more objects.

1.2 Objective

Given a list of sunlight exposure values corresponding each to a xy coordinate in 2D plane simulate tree growth based on how much sunlight each tree receives. The simulation should consider the occlusion of trees to calculate the amount of sunlight received by smaller trees. To achieve the growth rate each trees extent is set to 0.4 and the new calculated extent gradually increases as the simulation proceeds. Trees in areas with higher sunlight exposure grows faster and thus may occlude surrounding trees. These smaller trees in turn receives less sunlight and grow slower. To determine the new extents of trees we have set 10 layers corresponding to 10 extent ranges for trees to grow. As the layer increases the extent and the amount of sunlight received decreases. The GUI needs to render the tree growth as the results of the simulation occurs hence the rate of tree render needs to be faster than the simulation. The implementation of this should be done using parallelism and practicing safe concurrency principles to ensure thread safety. Included in this objective is to ensure thread safety in the java Swing library.

1.3 The need for thread safety

The synchronization of critical sections and object fields in our instance was important to coordinate thread access by the GUI and Simulation classes. These threads read and wrote concurrently to a shared memory location. Therefore the need for synchronization was essential to avoid data races. A more detailed account of synchronization will be explained in the following chapters.
Variables declared volatile aren't cached in hidden memory from other processes i.e. reading a volatile variable will always return the most recent change/write by any thread. Volatile variables are not implemented to lock anything. It is however more commonly used as a flag variable which is useful to control the flow of a functions by multiple threads. Declaring a Boolean variable volatile was useful for notifying threads when to function and when not to. This too will be further discussed in the upcoming chapters.

2 Class implementation

2.1 Land class

The initial land class contained no data structure or any method definitions. A two dimensional array was implemented to store all the sunlight exposure values via

the setFull method. The input file provided a xy coordinate in addition to the sun exposure value. Using these attributes each sun exposure value was indexed and stored sequentially into two arrays. The second array is used to reset the first array to their original values via the resetShade method.
The getter and setter methods were defined by returning the requested value. For both getFull and getShade each retrieves data from the original and the simulated array respectively. The setShade method on the other hand reduces a specific value in the sun exposure array to ten percent of its original value. The last method defined in the land class is the shadow method. By indexing all the values in the sun exposure array within a given trees extent we can reduce each value by setShade. The critical section for this method was synchronized to avoid any multiple read/writes.

2.2 Tree class

The main method defined in the tree class was the sunexposure which took a trees coverage of sunlight cells and calculated the average. A few conditional statements were included to ignore calculated indices which fell beyond the extent of the sun exposure array. Another condition was included to check for trees which had an extent less than one. The sungrow method in turn uses this average calculated to get the new extent. Since this was a write method, synchronization of the method was required to ensure two threads don't change the trees extent concurrently.

2.3 ForestPanel class

The only modifications made to Forest panel was the addition of volatile 'done' variable created in TreeGrow. Essentially it acts as a flag variable to indicate whether the thread should run or not i.e. paint or pause.

2.4 Simulation Recursive Action class

The Simulation class is extended by RecursiveAction. The divide and conquer approach was taken to operate on the trees array created in sunData. Setting a specified sequential cut off along with providing the sun exposure array, min and max extent range to the constructor, the trees array was divided recursively and each tree within a given range is operated on. The operations performed on each tree is the sungrow and shadow method which were previously discussed. These two operations were split between two simulation classes in order to maintain a sequential order of operation. The sungrow method is invoked first for each tree in a layer after which the shadow method is performed.

2.5 Shadow-Grow Runnable class

ShadowGrow is a runnable class which initiates the Simulation class. Within ShadowGrow are methods for pausing, resuming as well as running the Simulation class. The reason behind creating this was to control it as a thread from within the TreeGrow class. This class is also responsible for specifying the number of years/generations the simulation should run for as well as resetting the sunlight exposure and range to their original values after every ten passes. To implement the pausing and resumption of a thread we employ the thread methods 'wait' and 'notify'. On pressing pause the method suspend is called setting a variable 'suspended' to true. This is as a flag variable to indicate the wait method to be invoked. The thread is then woken up by the method notify which is called through the method resume and a second press of the play button.

2.6 TreeGrow class

The TreeGrow class acts as a controller since most of the GUI and user inputs is made through buttons defined by this class. The addition of four buttons as well as a text field to display the simulation year was incorporated. The four buttons play, pause, reset and end is discussed below.

1. Play – the play button either initiates a new simulation or resumes a paused simulation
2. Pause - Pausing sets the volatile variable done to true stopping forest panel from refreshing. It also suspends the ShadowGrow thread controlling the simulation
3. End – End simply quits the application GUI and ends the programming
4. Reset – the extent of all trees is set to 0.4 after which a volatile variable done is set false and a new forest panel thread is started to begin drawing the trees with new extents.

3 Programming models

3.1 Ensuring thread safety

To implement thread safety practices, we must ensure that every memory location in our program is either :

1. Thread local – only one thread ever accesses it
2. Immutable – its only read not written to Or
3. Synchronized – locks are used to avoid race conditions

For our program we've two data structures (trees and sun exposure arrays) which are both written to by multiple threads. Most getter and setter methods (for tree and land arrays) were synchronized as both threads for forest panel and for the simulation classes needed to access our arrays. Although Forest panel only needed to read from the trees array, the simulation class wrote new values to the tree objects extent fields. This situation is an example of concurrent read/write by multiple threads. By synchronizing the getter and setter fields of tree objects we can avoid data races since simultaneous read/writes were not possible. Within the simulation class recursive action splits the trees array among multiple threads. The pool of threads then operates on the trees by growing their extent and reducing the sun exposure values. Growing the extent is only completed by one thread at a time however this happens fast enough to be negligible. The second operation is to process the shade or new sun exposure values in the sun exposure array. The process is a large critical section under the shadow method which is synchronized as well. Here we may be worried that bad interleavings may occur due to threads acquiring locks non-deterministically especially in cases where trees occlude one another. The event of a thread getting the average of tree A and processing shade before another thread calculates the average of tree B which occludes A is highly likely. However, this isn't possible since both the sungrow and shadow method has been synchronized . The synchronization primitives put in place was ensure that liveness of the concurrent program was achieved in addition to avoiding situations of deadlock.
To inform threads of the program state a volatile boolean variable known as 'done' was initialized as true in the TreeGrow class. The purpose of the variable was to share state between the TreeGrow, Simulation and ForestPanel threads. The variable acted as a condition upon whether to render trees in the GUI.

3.2 Validation

Debugging a concurrent program isn't easy and may be futile if the first bugs only occur after a thousand iterations of the program. Forcing the program to produce errors is however possible. Validation of the program was conducted by printing essential variables such as volatile variable states, tree extents and sun exposure values to ensure the correct operations were being processed.

3.3 Model View Controller

The concept of MVC implies that each section of your program is responsible for a certain task. Some of the code controls the way the program works, some determines the appearance and other stores the data of the program. This approach makes things easier and more organized for other programmers to understand.
The model of our program is the data stores, namely tree and land classes. These classes are responsible for creating tree and land objects and any manipulation methods, getter and setters and defines the main components such shadow and sun exposure methods.
Our controller methods are simulation classes (ShadowGrow and Simulation) as well as the main class TreeGrow and SunData. These class dictate what should happen to our model and how to access functions such as growing trees via the play button or pausing simulation with pause. The sunData also controls where data is stored and the creation of tree and land objects
The view is the forest panel as this provides the visual aspect of the program. The view is defined by the GUI and how the user interacts with the application. Hence all buttons created in TreeGrow and the generation of trees created by simulation is part of the view.

4 Conclusion

Deciding on what to synchronize can be troublesome and time consuming if the programs functionality and theoretical nature of threads are not understood. In particular I found it hard to program a model which conformed to using consistent locks and using more fined grained locking. An in depth understanding of how threads operate also needed to be achieved as numerous errors occurred upon pausing and resuming threads. Using more locks could increase performance however at the expensive of threads needing to acquire locks and possibly deadlock.