CSC2002S Assignment 4

PHLALA005

Introduction:

This assignment aimed to create a typing game using threading and concurrency to manage different words falling simultaneously. Therefore it was important to access potential race conditions and prevent them

Code changes:

The major change was adding the FallingThread class. This class exclusively managed one "column" on the game board, moving the word down, resetting it when it reached the bottom and adding to score where necessary.

WordRecord was changed to add a "dropped" Boolean to indicate whether that word should be falling at that time or not. I also made some other small changes to help make the code run easier. Such as returning a boolean for whether the word hit the bottom when it's dropped.

WordPane was changed to repaint the words every 10ms to ensure that the words animated.

WordApp had a lot of functionality added. Code to make each button work was added, "dropping" the words on start, stopping them dropping and resetting on end and quitting the threads and quitting on quit. All the threads were started from within here and extra code was added to update the score as well as detect when the game was over (when the end button was not pressed).

Thread Safety:

Thread safety is about ensuring that threads are not interacting with the same resources simultaneously. For example if two threads are writing to the same place then it becomes a race and the output depends on which thread was faster. This is not predictable and leads to unexpected outcomes. This is what is considered unsafe behaviour, when the output is not predictable because threading was used. To ensure this didn't occur in my code, I reduced the number of points which this could occur by stopping the threads dropping the words before resetting them as well as a few other things where ordering was important. When accessing the same data but not changing it, thread safety does not apply. This is the example with WordDictionary.

Thread synchronisation:

All accessor and mutator methods within WordRecord and Score were changed to use synchronisation. This is because if two or more threads try to write to the same address space simultaneously then they can cause corruption. Therefore a "monitor" is established to control access to the resource so only one thread has access at a time. This prevents the issues described above.

Thread Liveness:

Thread liveness simply refers to ensuring that a thread is doing something consistently and not in a deadlock or similar situation where is it doing no processing but the system resources still need to be allocated to it. This is not a problem in this program because each thread is either completely inactive (i.e. waiting for the words to fall) or running in which the operation is simple and doesn’t require on any other external factors or threads.

Validation:

To ensure the code worked as planned, I tested a number of scenarios during gameplay to check that all the requirements of the game were met. Then I commented out chunks of code and reduced the wait time so the threads ran almost instantly to check against race conditions. The code held up and no race conditions were found.

Model View Controller:

In this game the view is WordPanel which controls the visuals of what is going on and provides information which the user can take action on. The controller is WordApp and FallingThead which both take user input and update the model to reflect that. The model is WordRecord and Score. These are edited by WordApp and FallingThread to reflect changes and then WordPanel takes these changes and updates the view. Therefore it is a Model View Controller system.

Github Link:

https://github.com/Lolarent000/CSC2002S-Prac-4