Synchronized Boids!

Homework #2, CSCI 322, Winter 2016

Due: Monday, February 8, at midnight.

Boids: This program is a continuation of the BOIDS program from the previous homework assignment. You will make revisions to the `boids02.rkt` program, resulting in `boids03.rkt`. Your new version should have all the features of `boids02.rkt`, plus the synchronization described below. If you haven't finished `boids02.rkt`, see me as soon as possible to work out the problems with your code.

Boid synchronization: The boids are not synchronized with each other. It is possible for one boid to loop many times more than the other boids. To prevent this we want to introduce a barrier so that all boids finish their force calculations before any advance to position updates, and all boids finish their position updates before any advance to calculating their force again. This will assure that all boids have the latest possible data about other boid positions when they do their calculations.

To solve this problem you will introduce a reusable two-phase barrier, as documented in listing 3.10 from *The Little Book of Semaphores*. The first turnstile will be after the force calculations and before the position update, the second turnstile will be after the position update and before looping back to calculate forces. (Feel free to rename the turnstiles, *etc.* to make the names more appropriate to this specific problem.)

As a first approximation to the solution to the synchronization problems, code up the two-phase barrier exactly as in the book (translated to Racket, of course). You can use an object for the barrier, or just use inline code, but the purpose of these parts of the code should be clearly documented in comments.

Reader-writer synchronization problems: Introducing the barrier raises a new synchronization problem. If you introduced the barrier and tested it (as above), it *probably* worked OK. This is a major problem with concurrent programming, many bugs lurk unseen, and it is difficult to construct test cases to make them appear.

The problem is that the reusable barrier run by each boid needs to know the number of boids, n. When a user clicks on the interface and creates a new boid, the value of n changes. This is a classic readers-writers problem. The boids are all readers, needing to read the value of n to run their barriers. The callback for a mouse click is a writer, needing to change the value of n, and should run in a thread so it can synchronize itself with the boids and with other writers.

Now, since there will generally be many boids but few mouse clicks, the potential for starvation has to be considered. But, we cannot use the writer-priority reader-writer solution from the book. We *don't* want the writer to write when only a portion of the boids are in the critical region, where they are reading n. For example, if there are 100 boids, and a writer tries to write when only 50 of them are in the critical region, then the writer will lock out the other 50 boids, and wait for the first 50 boids to finish their critical section. These boids, of course, cannot finish until the other 50 boids also finish, because they are using barrier code. Deadlock!

This probably wouldn't happen even if you implemented and ran the writer-priority reader-writer solution, since the boids spend the vast majority of their time calculating forces. That part is $O(n^2)$ while the rest is $O(n)$. Odds are very good that most mouse clicks will happen while calculating forces. But that doesn't mean we can assume it will *never* happen!

We need to think more carefully about the possibility of starvation, and at what points the writer can change n and introduce a new boid.

Solution idea: We need to assure that the writer does not change n at inappropriate times. To stop writers from writing, and to make sure only one writer writes at a time, let's assume the writers (the callback from the mouse click) use a mutex called `writeMutex`. Their code simply grabs the `writeMutex`, and also grabs the barrier mutex (why?), increments n, creates a new boid, and then releases the barrier mutex and `writeMutex`. Can we make all the other boids cooperate with this code, so that n is always consistent, but never starve the writer?

First of all, when only *some* of the boids are at the barrier, the other boids are finishing their processing. The boids at the barrier will wait for all others to finish, and then the last boid will close the previous turnstile and open the next one.

Second, when we introduce a new boid, we want it to participate with all the other boids in the barriers, and so it must start in the same *state* as (at least some) of the other boids. If we always start a boid at the top of its loop (calculating forces), then some of the other boids should also be calculating forces at the same time.

This "open enrollment" period for new boids *ends* when the *last* boid enters the *first* phase of the barrier. This would be a good time to stop any writers from adding a new boid. If we let this last boid (the one that is going to open and close the turnstiles) grab `writeMutex`, then writers will be locked out of writing while this boid and all the other boids do their turnstile thing. The boids (jointly) should hold onto `writeMutex` until they are all done using n.

After the first turnstile, the boids are updating their positions and they do not use n. If n were the only problem, we could let go of `writeMutex` for a while. However, we do not want the boids to release `writeMutex` yet, because then a writer could introduce a boid that was calculating forces while all other boids were updating positions, and one boid would be out of synch with all the other boids. n would also be a problem. Can you see why?

Instead, the boids should hold onto `writeMutex` until the last boid finishes the *second* phase of the barrier, and opens the second turnstile. At that point, boids begin calculating forces again, and a new boid can join the flock with no problems. So writers only write when boids are calculating forces, and the new boid would fit right in.

n can also be incremented at this time, so long as the *last* of the old boids has not entered the *first* phase of the barrier. But we already assured that that cannot happen, because the last boid grabs `writeMutex` immediately.

What about starvation? Is it possible for a writer to starve? See if you can convince yourself this can't happen, no matter how many boids are flying.

Assignment: Code up this solution idea in a program called `boids03.rkt`. There are some details left out (which mutex does the writer grab first? why? does it matter?). You should comment decisions you make and justify your code.