CS 361 - Concurrent Programming
Assignment 4

Dennis George

5/26/2021

1 Part 1 - Report

Description of classes and methods

1. Dog Class
   Dog is an abstract class and holds all of the shared data members for any of the actors in this simulation. All dog breed classes will derive from this Dog class. Dog also outlines some common methods described below:

   (a) *eat()* - sleeps the thread for a random amount of time.

   (b) *outside()* - sleeps the thread for a random amount of time. Simulates a dog doing whatever else they may be doing (sleeping, running around, etc...)

   (c) *run()* - defines the lifecycle of a dog of any breed. The lifecycle is defined as followed:

      i. *outside()*
      ii. *beginGetInLine()*
      iii. *endGetInLine()*
      iv. *beginEating()*
      v. *endEating()*

      The methods pertaining to getting in line and eating are all defined in the Kennel class.

2. DogBreed Class
   There is no actual class name *DogBreed*, however all distinct dog breeds in the problem will follow the exact same pattern. There are three concrete Dog classes:

   (a) Ridgeback

   (b) Coonhound

   (c) Doodle

   Each of these concrete classes only define a constructor, which will set the type of the dog, and start the thread. The *run()* method for the DogBreed class is defined in the abstract Dog class.

3. Kennel Class
   All of the synchronization and monitors will appear in this class. I am using the synchronize keyword for my methods instead of defining specific conditions. The kennel has a LinkedList called *queue*, which holds *Dog* objects. This queue is going to be read/written from in order to maintain order in the simulation. The Kennel class acts very similarly to the Database class in the reader/writer example from the course lectures. There are only four main methods in the Kennel Class which are outlined below:

   (a) *beginGetInLine()* - attempts to put a dog on the waiting queue for going in to the kennel to eat. As soon as the caller has access to the queue, they get added to the end. If another proccess is attemping to read from the queue, we do not allow the modification of the queue to happen, as it could lead to inaccurate results.

(b) *endGetInLine()* - when a Dog is successfully put on the queue, we can unblock any thread that was attempting to modify it. *notifyAll()* is called, and the platooning timestamp variable *startWaitingReadersTime* is set to be the age of the dog that was just added on the queue.

(c) *beginEating()* - when a dog is attemping to eat, we need to ensure that it is a valid moment for them. *wait()* is called, and the dog will block until they have access to the queue. The kennel then facilitates which dogs are allowed to eat, according to the specifications of the assignment (only one Doodle, multiple dogs of the same breed). Once we have the list of dogs that are allowed to eat, the kennel calls the *eat()* method on each dog.

(d) *endEating()* - call *notifyAll()* and unblock any dogs who were waiting to be proccessed.

Other things to note about the implementation: The amount of each breed of dog is configurable via the command line. See the makefile for details on what these parameters are.

Test cases:

1. Multiple Ridgebacks can eat together
   For this test case I was looking for output that indicated that there were multiple Ridgebacks waiting to eat, and are both successfully let in at the same time. If this works for Ridgebacks, then it is safe to assume that it also works for Coonhounds, as they have the same logic. In order to test this, I bumped up the number of Ridgebacks in the simulation.

```
Ridgeback0 is outside for 2 seconds
[Ridgeback@90e5243]
->Ridgeback3 is waiting to eat
[Ridgeback@90e5243, Ridgeback@fe756bb]
->Ridgeback1 is waiting to eat
->2 dogs of type Ridgeback currently eating
->Ridgeback3 is eating
->Ridgeback3 is done eating
->Ridgeback1 is eating
->Ridgeback1 is done eating
```

   The arrows here show that both Ridgeback3 and Ridgeback1 were successfully in the queue to eat. When the kennel was ready to serve them, it brought in both dogs, and the *eat()* method for both Ridgebacks was called.

2. Only one Doodle can eat at a time For this test case I was looking for output that indicated there were multiple Doodles in line, and that only one could get through to eat.

```
->[Doodle@203ff42a, Doodle@50e6d5b9, Coonhound@7f9e626e, Ridgeback@19ba421]
Ridgeback2 is waiting to eat
->1 dogs of type Doodle currently eating
->Doodle2 is eating
->Doodle2 is done eating
```

```
Ridgeback0 is outside for 2 seconds
->1 dogs of type Doodle currently eating
->Doodle0 is eating
->Doodle0 is done eating
```

This output shows a queue that has multiple Doodles in the beginning of the queue. When the kennel was ready to let dogs in, only one Doodle was allowed to eat, Doodle2. Once Doodle2 is finished eating, Doodle0 was allowed in to eat.

3. No breeds are mixed in the Kennel For this test case I was looking for output that indicated there were multiple breeds who were eligible to eat, but the kennel only allowed in one breed.

```
->[Coonhound@7f9e626e, Ridgeback@19ba421]
1 dogs of type Coonhound currently eating
->Coonhound2 is eating
->Coonhound2 is done eating
```

This output shows a queue that has two different breeds who are eligible to eat, however only the Coonhound gets through.

I believe that these test cases show that my implementation meets all of the turnstile requirements of the kennel:

1. Don't mix breeds, but allow multiple of the same kind in
2. Only allow one Doodle at a time for food aggression issues

Credits My solution is adapted from the Week8Part1 lecture slides, which demonstrates a Java implementation for database reader/writer solution using monitors.

2 Part 2 - Model Checking

1. Ben-Ari 4.12 (frog puzzle)
My solution is included in my submission under frog.pml. I used https://www.weizmann.ac.il/sci-tea/benari/sites/sci-tea.benari/files/uploads/keynotesAndEssays/invisible-slides.pdf as a resource in creating this script. Here is the output:

```
            ltl ltl_0: [] (! (((((((stones[0]==female)) && ((stones[1]==female))) && ((
      timeout
#processes: 7
                stones[0] = none
                stones[1] = male
                stones[2] = male
                stones[3] = male
                stones[4] = female
                stones[5] = female
                stones[6] = female
 37:    proc  6 (femaleFrog:1) frogs.pml:43 (state 11) <valid end state>
 37:    proc  5 (maleFrog:1) frogs.pml:21 (state 11) <valid end state>
 37:    proc  4 (femaleFrog:1) frogs.pml:43 (state 11) <valid end state>
 37:    proc  3 (maleFrog:1) frogs.pml:21 (state 11) <valid end state>
 37:    proc  2 (femaleFrog:1) frogs.pml:43 (state 11) <valid end state>
 37:    proc  1 (maleFrog:1) frogs.pml:21 (state 11) <valid end state>
 37:    proc  0 (:init::1) frogs.pml:79 (state 15) <valid end state>
7 processes created
```

Run the script in the command line using spin to verify the output. You can see that everything ends up with a valid end state and the ltl is passed.

The resource I found showed how to make a *proctype* for the maleFrog. I was able to come up with a similar implementation for the femaleFrog *proctype*. With both of these, I could create the *init* process which puts them together and runs the simulation.

2. Dining CS Students Problem My solution is included in my submission under student.pml. I didn't use any resources for this, and it was entirely adapted on my own using the Promela/Spin documentation.

   For the setup, I created 4 boolean arrays of size NUMSTUDENTS that represent whether or not a student is performing that operation. For example, the *eating* array of booleans, determines whether student at index *i* is eating or not. There is also three integer arrays that represent which student is using which item on the table. For the forks, there are NUMSTUDENTS entries, all initialized to -1. When a student picks up a fork, they put their index into the *forks* array. For *cards* and *phones*, there are only NUMSTUDENTS/2 slots, so we have to be more careful about inputting into these arrays. There are a few LTL's that are setup which will verify the safety and the fairness of my model. The *sem* LTL is setup to ensure that the value of the semaphore is always either 0 or 1 in sall states. This proves safety around the BOWLAMOUNT. The *fairn* LTL is setup to show that "eventually, always eaiting[n] is true". This proves that all students eventually get to eat.
   I setup a semaphore, using the atomic keyword and a count.
   With all of this setup, we can now go over the proctype of the student.
   The code itself outlines one proctype *student*, which takes the number that represents which student they are. This number is used to compute which index to the state arrays we should use. There are five main events in the *student* proctype:

   (a) Code
   Set *eating[i]* = false and *coding[i]* = true, atomically

   (b) Wait
   Set *coding[i]* = false and *waiting[i]* = true, atomically
   In this case, we need to try and get a left and right fork. First we try and get the left fork, and if it worked we try and get the right fork. If end up in the state where we try and grap the right fork first, then we wait to try and get the left fork.

(c) Eat
Set *waiting[i]* = false and *eating[i]* = true
In this state, we need to acquire the semaphore in order to decrement the amount of food in the bowl. If after this happens, the amount of food in the bowl gets below 1, then we transition to the Order state. Otherwise, atomically set *eating[i]* = false, and transition to the Done state.

(d) Order
Set *ordering[i]* = *true* and *eating[i]* = false.
If you are the even student, then you should try and get the phone that is on the left and the card that is on the right. If you are an odd student, you are doing exactly the opposite. Once we have a phone and a card, we can increment the BOWLAMOUNT back to being 10 - its original value. Atomically set *ordering[i]* = false, and reset all of the state values of *phone* and *card*.

(e) Done
Release the semaphore, as we are no longer changing BOWLAMOUNT in either Eat or Order state. Return back to the Code state.

Run the script in the command line using spin to verify the output. NOTE: You will see that there are some unreached statements for some of my ltl claims. This is just a warning, as it cannot reach the end state, due to there not being one set up. The problem doesn't specify when to terminate, so I do not terminate.