CSC2002S Aidan de Nobrega DNBAID001

Assignment 2: Concurrency – Concise Report 24/08/2015

Modifications

I initially started with the skeleton code that was provided to us, but I quickly realised that it would be easier to maintain thread safety if I redesigned the program from scratch with thread safety in mind. In some cases, I copied the code over from the skeleton program where it was simple enough and had no real bearing on the concurrency of the threads. Exception handling has been implemented where necessary.

In the `GolfBall` class, the only change I made was a minor tweak to the way objects are instantiated. There is no longer a method to increment the value of the `numBalls` static variable. The value is merely incremented in-line when the identifier for each ball is set.

In the `DrivingRangeApp` class, I introduced a new global `AtomicBoolean` variable, `holding`, which is passed to all relevant classes via the constructors for those classes in the same way as `done`. This variable indicates whether Bollie is in the interim phase between having collected balls from the field and adding them to the central supply. This is a safe implementation because only one instance of `BallStash`, `Range`, and `Bollie` should exist for each driving range. The `sizeBucket` and `sizeStash` static variables in `BallStash` are now instantiated via the constructor for the class as well (instead of calling mutators from `DrivingRangeApp`). I also added some randomisation generation for the length of time over which the driving range is open and I added a single synchronisation block which will be explained in the next section.

In `BallStash`, I added an `AtomicInteger` – `ballsInStash` – to keep track of the number of balls in the stash at any one time as well as an accessor for this variable. I added the `fillBucket()` method – a function which returns an array of `GolfBall` objects, the size of which is always the value of `sizeBucket` – and `replenish()` – a function which simulates `Bollie` adding collected balls to the central supply. I added synchronisation to both these nonstatic methods.

In `Bollie`, I added some randomisation to the timing of Bollie's actions and I added the relevant invocation of `collectBalls()` in `Range` and `replenish()` in `BallStash`.

In `Range`, I added the methods `collectBalls()` and `hitBall()`. The first is called from `Bollie` (as described above) and the second is called by `Golfer` objects. They both return void and are synchronised on the calling object.

In `Golfer`, I added a `boolean` – `goHome` – which is a flag that drops when the golfers are no longer allowed to fill their buckets because the range is closed. The reason for having this variable in addition to `done` will be explained in the next section. It is not necessary for this variable to be anything but a primitive. I moved the console output from this class's `run()` to the relevant methods in BallStash and Range. Invocation of `fillBucket()` and `hitBall()` are made here with respect to swing times that are randomly generated for each instance of this class.

Concurrency and Synchronisation

The ball objects are passed from collection to collection as they are dispensed to the golfers, hit onto the field, and fetched by Bollie. An `ArrayBlockingQueue` is used to represent the central stash because it comes with the built-in functionality of making consumer threads wait until there are items in the queue before dispensing. I used `AtomicBooleans` for condition variables that were shared between threads because `AtomicBooleans` come with the ability to execute read-modify-write and check-then-update compound actions atomically. This reduces the risk of incurring bad interleaving race conditions because no intermediate states can be accessed while these values are changing.

When a `Golfer` thread attempt to fill its bucket, it invokes `fillBucket()` in `BallStash`. This method acquires the central stash as a lock. When the method starts executing, it makes the initial check of ensuring the driving range is still open. If it is open, the method checks to see whether there are enough balls in the `BlockingQueue` to fill a bucket. While this is not true, the thread calls `wait()` and releases the lock, allowing other golfer threads to acquire the lock and allowing Bollie to add balls to the stash (because `replenish()` also requires the central stash lock). When Bollie has filled the central stash, he calls `notify()` and releases the lock, thus allowing golfer threads to acquire it and fill their buckets (if there are enough balls). Initially, I had a spin-wait, but using `wait()` and `notify()` is a lot more efficient. When the thread is notified, it does an additional check to ensure the range is still open. If this method returns an empty bucket because the range closed during the process, the `goHome` flag is dropped and the golfer knows to return his empty bucket and go home. `replenish()` also has to acquire the lock to ensure that there is not race condition between the producer (Bollie) and consumers (golfers) accessing the stash. The method checks `holding` to see whether Bollie has actually undertaken a collection before allowing him to add anything to the stash. Without the `holding` variable, Bollie can attempt to replenish the central supply without collecting balls first after the range has closed.

There is no thread safety to ensure in the Bollie class, but it is important to note that the balls he has collected but not yet added to the central supply are held in a `BlockingQueue` as well.

For a golfer to hit a single ball onto the field, the method `hitBall()` has to acquire the range as a lock. However, when the Bollie class calls `collectBalls()`, the lock is acquired preferentially and the golfers cannot swing. Because of this simple mutual exclusion with fine grained synchronisation (on each swing), it is unnecessary to have the `AtomicBoolean`, `cartOnField`, at all because Bollie always gets preference and the golfers have to wait on their next swing for Bollie to release the lock. When Bollie has collected balls, his holding flag is raised and once he has added them to the supply, it drops.

Golfers have the `goHome` variable in addition to done because if a golfer thread has been waiting in the `fillBucket()` method, it is likely we can experience a visibility error such that the thread misses the done flag's change of state.

The last two lines in `DrivingRangeApp` are contained in a synchronised block to ensure that no code from `BallStash` attempts to execute in between the flag dropping and the line of console output indicating as such.

Validation

To ensure that no deadlock was possible, I designed the program such that no one thread ever holds more than one lock at a time. This makes it impossible to deadlock. All shared variables are updated atomically and methods are locked such that threads do not ever access stale data.

There is no canonical way to debug a program to expose race conditions. But my program was designed from the bottom up with concurrency in mind and no step was undertaken without assessing what data races might exist or what intermediate states might be exposed. However, I did create a test harness that executed the program with different starting variables hundreds of times and wrote the output to file. Then I ran a FileChecker class over all the files and the FileChecker would report if any of the files failed a certain invariant. For example, whenever the FileChecker came across the line in the output where Bollie starts collecting, it would make sure the next line it read was either Bollie finishing his collection or the range closing. Another example of an invariant checked by the program was any golfer hitting more balls than a single bucket's worth or Bollie beginning a collection after the range has closed. I checked the program in different extreme states, such as with no golfers or balls, a bucket capacity of 0, the possibility of having fewer balls than are required to fill a single bucket, etc.

Extensions & Extra Credit

I added the option to execute the program in two modes. The first is the standard program which can be used to test the concurrency and basic assignment prerequisites. The second is with the added functionality that allows golfers to arrive at the driving range at randomly-generated intervals with a predetermined number of buckets to play (also a randomised value). Each golfer's arrival is announced along with his intended number of buckets and he goes home when he has played all his buckets. This implementation required me to create the class `GolferGenerator` which also extends `Thread` and creates `Golfer` threads as needed. You will note that, although golfers cannot hit balls onto the field while Bollie is collecting, there will still be output statements announcing golfers' arrivals and departures during this period.

In addition to this, my code is thoroughly commented and javadocced. Almost all of the implementation details can be gleaned from the comments themselves.