REPORT

MNGKHO012

Concurrency Assignment: Medley Simulation

Task

The medley simulation program mimics a relay-style swimming race where multiple teams compete. We were given a program that was not functioning correctly and had multiple concurrency issues. The goal of this assignment was to identify and correct these issues by applying appropriate synchronization mechanisms to ensure thread safety and prevent race conditions. Additionally, some improvements and extensions were made to enhance the simulation's behavior and performance.

Concurrency Issues Highlighted

1. **No Synchronization** - None of the classes conformed to the Java Monitor Pattern and none of the fields in the classes, especially classes that are accessed by multiple threads had atomicity. This means that they were not thread safe and prone to a plethora of data races and race conditions, such as multiple swimmers occupying a grid block at once.

2. **StadiumGrid.enterStadium()** method has an incorrect condition in its while loop, which was impacting its functionality by failing to check if entrance was free correctly.

3. **StadiumGrid's busy waiting** – methods such as enterStadium(), jumpTo() and moveTowards() were all using a busy waiting algorithm to check if grid blocks were available. This approach although correct, is inefficient because it involves constant checks in a loop and was sometimes working incorrectly.

4. Swimmers did not enter the race in the correct order of their swim stroke. There were no mechanisms to enforce the required order in the SwimTeam or Swimmer classes.

5. Swimmers started swimming at random without any synchronization or rules in place. This lack of enforcement led to chaotic race starts.

6. Although not a concurrency issue, the start button was non-functional. The program was expected to control the race start but failed to do so, impacting the simulation's overall performance.

Improvements made to each class

1. **MedleySimulation**
   - Added a start signal, using an *AtomicBoolean*, and then added functionality in the start button, so that when the start button is clicked the that are waiting are signaled to start the simulation.
   - Added a betting feature that allows the user to place bets on which team is going to win, at the end of the race they are notified if they won or were close.
2. **GridBlock**
   - Replaced *int* with *AtomicInteger* to ensure thread-safe updates to the *isOccupied* field.
   - Made methods synchronized to ensure mutual exclusion when accessing or modifying the block's state. Added *notifyAll() to* inform waiting threads when the block is free.
3. **StadiumGrid**
   - Added CountDownLatch for synchronizing the start of the race.
   - Added synchronization on the entrance block to ensure that only one person can enter at a time. This included the use of a synchronized block, locking on the *entrance* block and using *notify* and *wait()*.
   - Added synchronization on the newBlock and currentBlock to manage block occupancy safely. This included the use of a synchronized block, locking on the blocks and using *notify* and *wait()*.
   - Added synchronization on the *newBlock* and *currentBlock* similar to *moveTowards()* to ensure safe block transitions.

4. **Swimmer**
   - Added synchronization to ensure only one swimmer from a team swims at a time, using synchronized blocks and stadium.wait() and stadium.notify().
   - Used CountDownLatch to wait until all teams are ready before starting the race.
   - Used synchronized blocks to manage the start and end of a swimmers racing.
5. **SwimTeam**
   - Added synchronization to ensure each swimmer thread starts properly. The synchronized block and wait() method are used to wait for each swimmer to signal they are ready before starting the next swimmer.
6. **PeopleLocation**
   - All getter and setter methods that modify or access shared resources (inStadium, arrived, and location) have been made synchronized. This ensures that multiple threads can safely interact with these fields without causing race conditions.
7. **FinishCounter**
   - Replaced the boolean flag with AtomicBoolean, which allows atomic operations and thread-safe updates without explicit synchronization.
   - Used an ArrayList to store the top 3 teams to cross the line, so we can track second place and third place.

8. **CounterDisplay**
   - Now displays the results for the 1st, 2nd and 3rd place teams.

Conclusion

Working with multiple threads and shared resources is very delicate in nature, it requires a lot of careful consideration of synchronization. It is no secret that synchronization, especially too much of it has some cost on the performance, however I soon realized that the lack of synchronization has a far greater cost. I observed many times during the assignment that lack of proper synchronization can lead to peculiar, unpredictable and outright incorrect behaviors that degrade the quality of the program. These race conditions were so disgusting I even considered synchronizing every line.

In this assignment, synchronized blocks ensured that only one thread could access shared resources like grid blocks at a time, preventing data races. The CountDownLatch synchronized race starts, ensuring fairness. Finally, using notify() and release mechanisms helped manage block acquisition and release, keeping threads coordinated and avoiding deadlocks.

Proper synchronization, atomicity, and thread safety are not just fancy features but are essential elements that ensure a multithreaded/concurrent program function correctly and reliably. The improvements made in this assignment highlights why thread safety is absolutely necessary for a concurrent program of this nature to consistently function holistically and provide correct results without race conditions or data races.