University of Cape Town

CSC2002S Club Simulation

Will Madikizela
MDKWIL001

Table of Content

INTRODUCTION

In this assignment, the objective is to enhance and correct a multithreaded Java simulation of patrons inside a club. The primary goal is to implement synchronization mechanisms that guarantee the simulation's compliance with specified synchronization constraints, ensuring both safety and liveness. The simulation commences when the user initiates it using the Start button. As the simulation progresses, patrons enter the club through the entrance door, proceed to the bar area, partake in various activities such as dancing and wandering, and eventually decide to depart through the exit. The key challenge is to identify and rectify any concurrency issues within the existing codebase to ensure that all defined behavioral rules are consistently upheld, eliminating potential hazards and deadlocks while maintaining a smooth and continuous flow of the simulation. By successfully addressing these concurrency concerns, the final Java simulation will provide a realistic depiction of a club environment, accurately reflecting the movements and interactions of the patrons in a safe and efficient manner.

JAVA CONCURRENCY FEATURES

The use of Java concurrent features through the **AndreBarman** class will be demonstrated in the example that follows. It's vital to notice that comparable concurrent mechanisms are used across different classes in order to show their practical use in a concise manner. This strategy seeks to preserve the report's succinctness while guaranteeing a thorough comprehension of how these features are constantly used throughout the simulation.

AndreBarman.class
AndreBarman, incorporates various Java concurrency features to manage the behaviour of the barman in a club simulation.

1. Synchronized Methods:

   The **synchronized** keyword is applied to the **personToBeServed()** method. This ensures that only one thread can access and execute this method at a time, preventing potential race conditions when checking if a person needs to be served.

2. Synchronized Blocks

   Synchronized blocks are used within the **moveBarmanAcross()** and **checkGamePaused**() methods. A synchronized block is created using an object as a lock to ensure exclusive access to the critical section of code enclosed within the block. In **moveBarmanAcross**(), the **currentBlock** is used as the lock to ensure that only one thread can manipulate it at a

time. In **checkGamePaused**(), the **paused** object is used as a lock to ensure proper synchronization when checking if the game is paused.

3. Thread Sleep:
   The **Thread.sleep()** method is used in the **servingPeople**() method to simulate a delay when the barman serves drinks. While this isn't a concurrency mechanism, it's important to note that this method will pause the execution of the current thread for a specified duration.

4. AtomicBoolean:
   An AtomicBoolean named **paused** is used to manage the paused state of the simulation. AtomicBooleans provides atomic operations on boolean values, ensuring that multiple threads can safely read and modify the value without encountering race conditions.

5. CountDownLatch:
   A **CountDownLatch** named **startSignal** is utilized to coordinate the start of the simulation. The **run**() method waits for the **startSignal** to count down to zero before commencing the barman's movements. This mechanism ensures that all necessary setup tasks are completed before the simulation begins.

6. Thread Execution and Interruption:
   The **run**() method of the **AndreBarman** class is where the main execution logic of the thread resides. It starts with the **startSimulation**() method to wait for the simulation to start, then it repeatedly checks for the game being paused using **checkGamePaused**(). The **moveBarmanAcross**() method is called to handle the barman's movement, and the loop continues until the serving is complete. Interruption handling through **InterruptedException** is utilized to properly manage thread interruption scenarios.

This shows how the classes effectively utilized synchronized methods, synchronized blocks, atomic variables, CountDownLatch, and thread sleep to manage the concurrent behaviour of the barman in the club simulation. These concurrency mechanisms ensure safe and synchronized access to shared resources, proper coordination among threads, and controlled execution of the simulation's components.

THREAD SAFETY

Thread Safety

Thread safety was ensured by ensuring that threads do not access the same shared memory simultaneously, To prevent such actions from occurring, the methods were synchronized. This allows the thread to only access the memory when the lock is released or never acquired. There is no possible risk of deadlocks since the synchronized methods were never nested. A class is thread safe when multiple threads can access it and still operate smoothly. If data is accessed by multiple threads, it should be contently protected. The reason why data needs to be protected is that multiple threads can access the same shared memory simultaneously. Threads may read the updated data while they need the data from memory. Synchronization helps prevent multiple threads from accessing the same shared memory.

Thread Synchronization

Thread synchronization is very important to prevent threads from accessing the same shared memory, resulting in deadlocks and corrupted memory. All the getters and setters that are accessed by multiple threads in the program are synchronized to ensure that only one thread reads or accesses the data at a time. If these methods were not synchronized, some threads would end up reading and displaying the wrong values.

Liveliness

This simply refers to whether the program can complete tasks in a reasonable amount of time. Since shared memory can only be accessed by one thread at a time, does our program update in time? Yes, the performance is quite fast and updates in time. The use of Java concurrent features allowed us to have good performance.

No deadlocks

Deadlock is a condition where threads are waiting for each other, this is mostly common when more than one deadlock is required. In our program, all methods accessed by multiple threads are synchronized to ensure that we do not have any deadlocks.

SYSTEM VALIDATION

I validated my program by changing the getter and setter from synchronization to just a normal setter and getter method. This showed me that threads were accessing the same method at the same time. Thus, this validated that the setter and getter methods needed to be synchronized.
I also changed all the volatile variables to normal variables, and I experienced an abnormal variable update. This proved that these variables need to be accessed by one thread at a time when running the program.
I checked the program by running it multiple times, and it was bug-free. This also proved that our programme is race condition-free. The game did when ran multiple times had no errors showing, this also proved that our program is deadlock-free.

CONCLUSION

The simulation impeccably upholds the prescribed rules, meticulously ensuring adherence to the defined behavioral constraints. The principle of liveness is rigorously safeguarded, fostering an environment where tasks are executed promptly and patrons' actions are promptly reflected. Moreover, the specter of deadlocks is effectively averted through a discerning synchronization strategy. This strategy focuses solely on the pivotal methods, thereby minimizing the potential for simultaneous access conflicts. Notably, strategic placement of `wait()` and `notify()` blocks at critical junctures guarantees that patrons remain highly responsive without falling into the quagmire of perpetual waiting cycles.