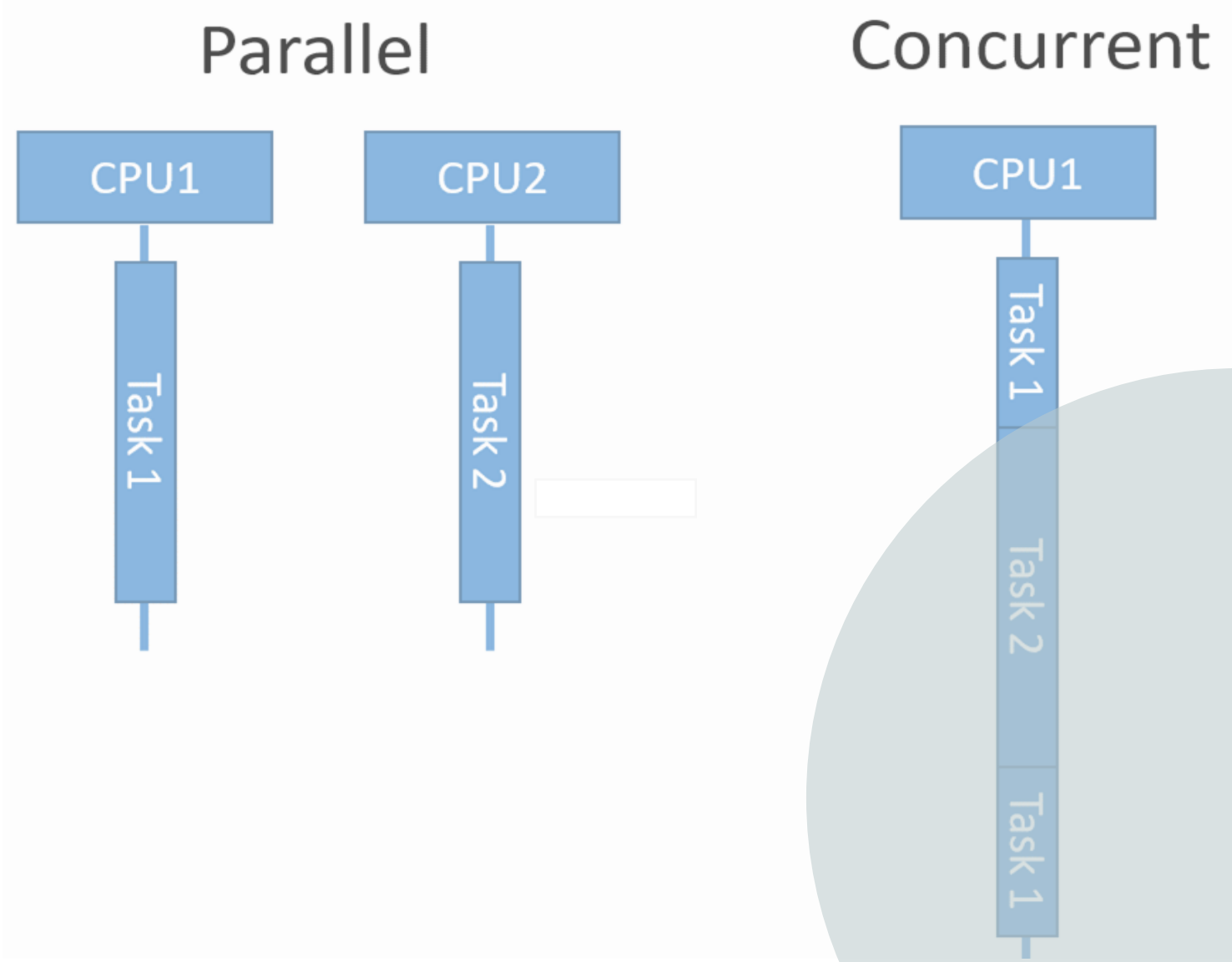

concurrency

By: Tokelo Makoloane

MKLTOK002

CSC2002S

INTRODUCTION

In this project, I've edited a club simulation that models the behavior of patrons inside a club environment. To ensure the correctness and safety of the simulation, I've implemented synchronization mechanisms in critical sections of the code where multiple threads interact.

CLASSES EDITED AND DESCRIPTION

- **ClubSimulation**: This class is the main driver of the simulation. It initializes the simulation parameters, creates threads for patrons, and sets up the GUI. The key methods here are **setupGUI** and **main**.
    - Firstly, the start and pause button implementations are located in this class and how I edited them to do their specific actions is that; for the start button I moved every instance of the thread class/package to the method of the action listener for the start button, therefore now the every thread of the game will not start until a user press start, but the GUI of the game will still be visible.
        - For the pause button: I implemented using combination of a flag ("***isPause***") and an object lock ("***pauseLock***"). *isPaused* is a volatile boolean variable that indicates whether the simulation is currently paused or running. When isPaused is true, it means the simulation is paused. When it's false, the simulation is running normally.

          pauseLock is an object used as a lock to coordinate the pause and resume actions among threads. It's a simple object used for synchronization purposes. Threads can wait on this object using ***pauseLock.wait()*** and can be notified to resume using ***pauseLock.notifyAll().***

          When a user clicks the "Pause" button, the *isPaused* flag is toggled to true. This indicates that the simulation should be paused. Threads within the simulation periodically check the value of *isPaused* to see if they need to pause their actions.

Within each thread's code that requires pausing, there's a section where it checks the value of isPaused. If it's true, the thread knows it needs to pause.

After checking the flag, if the thread determines that it needs to pause, it enters a ***synchronized*** block where it waits on the *pauseLock* using ***pauseLock.wait().***

This causes the thread to wait until another thread (likely the UI thread handling the button click) notifies it using ***pauseLock.notifyAll().***

When the "Resume" button is clicked and the isPaused flag is set to false, the UI thread that handles the button click uses pauseLock.notifyAll() to notify all the waiting threads that they can resume.

In summary, the pause mechanism uses the *isPaused* flag to signal to threads that they should pause their actions. The *pauseLock* object is used to coordinate the actual pausing and resuming of the threads. Threads that need to pause wait on this lock, and when the simulation is resumed, the waiting threads are notified and can continue their execution.

This approach ensures that threads pause and resume gracefully without interfering with each other, maintaining the overall simulation's correctness and responsiveness.

- **ClubGrid**: This class represents the grid layout of the club. It handles the movement of patrons, their entry, and exit from the club. It includes methods like **enterClub**, **leaveClub**, and **move**.
    - **The enterClub:** method controls the process of a patron entering the club. It is synchronized on the instance of ClubGrid to ensure that multiple patrons cannot simultaneously enter the club and exceed its capacity limit.

      The while loop checks two conditions: whether the maximum number of patrons (*ClubSimulation.max*) inside the club has been reached, and whether the entrance is occupied.

      If either of these conditions is met, the thread waits using ***wait().*** This prevents a patron from entering the club if it's full or the entrance is currently occupied.

      Once the conditions are met for the patron to enter, the entrance lock (*entranceLock*) is acquired to ensure exclusive access to the entrance block.

The patron's arrival is registered by incrementing the peopleArrived counter.

The patron acquires the entrance block using the get method of the entrance block, and this block becomes their current location.

The counters for people inside and the patron's location are updated, and the entrance lock is released.

- **The leaveClub:** method handles the process of a patron leaving the club. It is synchronized on the ClubGrid instance to ensure proper coordination among patrons leaving the club and managing the count of people inside.

  The method first acquires a specific exit lock (exitDoor[0]) using lock() to ensure that only one patron can leave through the exit at a time. The patron's current block is released using the release method, making it available for others to use.

  The counter for people inside the club is decremented, indicating that one patron has left. The patron's location status is updated to reflect that they are no longer inside the club.

  The *ClubSimulation.currentInside* counter is decremented, and a ***notifyAll()*** is called within a synchronized block to notify waiting patrons that there is now space available for them to enter.

The purpose of the array of locks **(exitDoor**) is to prevent potential deadlocks and ensure proper synchronization when multiple patrons are trying to leave the club through different exit doors simultaneously.

Consider the scenario where multiple patrons are trying to leave the club through different exit doors simultaneously. If there is only one lock for the exit door, and each patron tries to acquire that lock, they might end up waiting for each other indefinitely, leading to a deadlock.

To avoid this situation, an array of locks (exitDoor) is used, where each lock corresponds to a specific exit door. This ensures that patrons leaving through different exit doors can acquire their respective locks without conflicting with each other. As a result, the potential for deadlocks is significantly reduced.

In summary, the **enterClub** and **leaveClub** methods of the **ClubGrid** class ensure that patrons can enter the club while respecting capacity limits and that they can leave while properly updating counters and releasing resources. The synchronization mechanisms used in these methods guarantee that the club simulation runs smoothly and adheres to the behavioral rules defined in the project specifications.

- **PeopleCounter** - *personArrived method*: The personArrived method is synchronized to ensure that the number of people waiting outside doesn't exceed the club's capacity. This synchronization prevents race conditions between the threads trying to update the peopleOutSide counter.

  PeopleCounter - *personEntered method*: The personEntered method is synchronized to make sure that the counter for people waiting outside (peopleOutSide) is decremented atomically. It also checks whether any waiting patrons should be considered as entering the club based on the available space.

  The **startWaitingIncrement** method in the **PeopleCounter** class is responsible for continuously incrementing the count of people waiting outside the club. This method is used to simulate the arrival of new patrons outside the club over time. Let's break down how this method works:

  - The method startWaitingIncrement creates a new Thread object called waitingIncrementThread. This thread will run a loop that continuously increments the count of people waiting outside the club.
  - Inside the waitingIncrementThread's loop: The thread is put to sleep for 1000 milliseconds (1 second) using *Thread.sleep(1000)*. This sleep time is used to simulate the arrival of new patrons over time.
  - After sleep, the *personArrived()* method is called. This method is responsible for incrementing the count of people waiting outside the club.
  - The loop runs indefinitely as indicated by while (true). If an InterruptedException occurs during sleep (possibly due to the thread being interrupted by an external action), the thread's interruption flag is set using Thread.currentThread().interrupt(), and the loop is terminated using break.

    Finally, the waitingIncrementThread is started using waitingIncrementThread.start(), which begins the continuous incrementing process.

  In summary, the **startWaitingIncrement** method creates a new thread that runs an infinite loop. Inside the loop, the thread sleeps for 1 second to simulate the arrival of new patrons and then increments the count of people waiting outside the club using the personArrived() method. This continuous incrementing process helps maintain a realistic simulation of patrons arriving at the club over time.

And that concludes how I enforced the simulation rules.

CHALLENGES FACED

1. **Concurrency Control**: Managing concurrent access to shared resources like the entrance, exit, and club counters was challenging. Ensuring synchronized access without causing deadlocks or compromising liveness was a significant challenge. Mostly was when I made the pause/resume button to only pause/resume the patron's threads but not to pause every threads of the simulation.
2. **Thread Coordination**: Coordinating threads for proper execution, pausing, and resuming required careful design to prevent race conditions and unexpected behaviours.
3. **Liveness and Deadlocks**: Ensuring liveness (no thread starvation) while preventing deadlocks was a delicate balance. Improper synchronization could lead to situations where threads became stuck or starved.
4. **Count peopleOutSide**: It was challenging to implement the ***startWaitingIncrement*** method to have a different thread that will run without disturbing all of the other threads of the simulation and also to have a specific coordination with the thread that is responsible for the patrons inside the club and the ones who left the club.

ENSURING LIVENESS AND PREVENTING DEADLOCKS

- **Preventing Deadlocks:**
    - *Lock Ordering*: To prevent potential deadlocks, a consistent lock ordering principle is followed throughout the code. This ensures that threads acquire locks in a predictable order and avoid circular dependencies. Like when Patrons was walking in the club one by one than all clustering on the entrance block at once and not patrons getting inside the club.
    - *Release Before Wait*: In cases where multiple locks are involved, locks are released before waiting. This approach prevents situations where one thread holds a lock while waiting for another, leading to a deadlock.
    - *Timeouts***:** Using lock acquisition with timeouts. Instead of waiting indefinitely, a thread can wait for a specified amount of time to acquire a lock. Like in the code where there were continuous time outs for threads of patrons and the ones for counting patrons inside, outside and the ones who left.

- **Ensuring Liveness:**
    - *Entrance Lock*: The entranceLock ensures that patrons do not wait indefinitely at the entrance. If the club is full, patrons are blocked until space becomes available, preventing thread starvation.
    - *Exit Locks*: The exitDoor array of locks ensures that patrons leaving the club do not deadlock. Each exit door has its lock, avoiding contention and potential deadlock scenarios.
    - *Wait and Notify*: The wait and notifyAll mechanisms are used appropriately, ensuring that threads waiting for entry, exit, or pausing can resume without getting stuck**.**

LEARSONS LEARNED

1. **Selective Synchronization**: Applying synchronization only where necessary helps in reducing contention and improving performance. Not all methods need synchronization, and careful consideration is required to identify critical sections.
2. **Testing and Debugging**: Thorough testing under various scenarios helped identify synchronization issues. Debugging concurrent code requires a deep understanding of thread interactions and careful analysis of logs.
3. **Concurrency Patterns**: Implementing concurrent systems requires understanding patterns like locking, waiting, and signalling. Proper usage of these patterns ensures effective synchronization.
4. **Concurrency Complexity:** Working with concurrent programs can introduce subtle bugs and challenges that are not present in sequential programming. It was crucial to thoroughly understand the concepts of synchronization, locks, and thread coordination to ensure correct behavior and avoid issues like data races and deadlocks.

CONCLUSION

In conclusion, the synchronization mechanisms implemented in each class are well-suited to their specific requirements. They ensure that the simulation adheres to the specified rules, preventing race conditions, enabling liveness, and avoiding deadlocks. The design approach and the lessons learned highlight the importance of selective synchronization, proper testing, and understanding concurrency patterns in developing effective concurrent systems.

REFERRENCE

- **ScienceDirect:**

  https://www.sciencedirect.com/science/article/pii/S0004370222001461

- **DiVA-Portal:**

  https://www.diva-portal.org/smash/get/diva2:1464542/FULLTEXT01.pdf

- **Jenjov.com:**

  https://jenkov.com/tutorials/java-concurrency/deadlock-prevention.html

- **Guru99:**

  https://www.guru99.com/dbms-concurrency-control.html

- **GeeksforGeeks:**

  https://www.geeksforgeeks.org/concurrency-control-techniques/

  https://www.geeksforgeeks.org/introduction-to-timestamp-and-deadlock-prevention-schemes-in-dbms/#:~:text=Timestamp%2Dbased%20concurrency%20control%20and,the%20execution%20of%20concurrent%20transactions.