Report SEC

Explanation of My Design:
The two classes that use threading in my design is RoboManager Class and Map Class.

RoboManager Class
The RoboManager Class is responsible for creating Robots every 1500 milliseconds on a thread from the given thread pool. It's other responsibilities are managing Robot to Robot interactions i.e. whether the Robot can move to a particular grid space without crashing with another robot.

- Thread Responsibilities
  Worker threads in the thread pool created by executor are responsible for running individual instances of the Robot class concurrently. Each Robot instance represents a separate robot with its own behaviour

- Thread Communication
  These threads that run Robot instances don't directly communicate with each other. Instead it avoids crashing into each other by communicating with a shared 2D array.

- Thread Resource Sharing
  All robots share an 2D array and a ConcurrentHashMap. To avoid race conditions any code that checks it or edits it is in Map Class, except for RoboManager Class. This class makes sure that only one Robot instance is checking the map at a time to determine the Robot's next move. In my design one Robot is is allowed to check the map in RoboManager, it f finds a space that isn't occupied by another Robot and gets the move validated by the Map Class

  With the ConcurrentHashMap add and iterating through functions are automatically handled by it.

- Thread Termination:
  - The threads created by the executor are managed and terminated by the executor itself. When executor.shutdown() is called, it initiates an orderly shutdown of the executor and its worker threads, ensuring that all submitted tasks (in this case, Robot instances) are completed before shutting down. This happens when GameOStatus is set to true. Before starting the loop to create a Robot it checks this status. Newly created threads take a few milliseconds to realise they need to stop.

In summary, my design uses a thread pool managed by ExecutorService to run multiple Robot instances concurrently. These threads access shared resources in a coordinated manner using synchronised blocks to avoid race conditions. The threads are terminated gracefully by shutting down the executor when the game is over.

Map class
The map class coordinates movements with the game logic. It also keeps track of the score and figures out when the game is over. Incorporates key game logic and interactions. Wall creation also happens here.

- Thread Responsibilities
  There are two threads running in Map Class.
  1. timeScorer
     This thread is responsible for adding +10 points to the GUI every one second and updating the score label on the UI using Platform.runLater().
     When a Robot reaches (4,4) this thread is interrupted.

  2. wallExecute
     This thread is responsible for taking Wall coordinates from the blocking queue and trying to create a wall with them by calling createWall(). If createWall doesn't happen due to something occupying the area it returns false and the next Wall coordinate is taken till a wall is created or waits till another Wall coordinate is entered. It also communicates with the UI through Platform.runLater() to update the queue label (q) on the UI.

  3. Robot Threads call functions in this class to check if their move is ok and then move to that square

- Thread Communication + Thread Resource Sharing
  The UI thread, every time a part of the arena is clicked, calls addWall(int x, int y) in Map Class. What this function does is add the coordinates given to a Queue effectively allowing the UI thread to communicate with the wallExecute. wallExecute will take the coordinate from the queue and carry out its responsibilities

  Another one is the int score variable. This is accessed by two functions; timeScore() and roboMove. Hence before these methods change or access the score variable they need to get the mutex - scoreMutex to avoid race conditions. These two methods are run by different threads

  Last bit of thread communication is between createWall, roboMove(), roboMoveHorizontal() and roboMoveVertical. These methods access the 2D map to edit where a robot is moving to or is at, also where the w is being created if its been broken. Hence every time these methods access the 2D array for reading or editing it is only after t they get a mutex. Different threads edit the 2D array.

  Threads share resources such as the map array, score, noOfWall, and the queue. Access to these shared resources is protected by synchronised blocks (synchronised(mutex) and synchronised(scoreMutex)) to prevent race conditions. The map array is shared among

threads to represent the state of the game board. Synchronisation ensures that multiple threads can access and modify the map safely.

- Thread Termination:
  The wallExecute thread is terminated by calling wallExecute.interrupt(). This done when roboMove() realises that a robot is about to enter citadel. The thread does loop for every when looking for the next wall after the wall it was supposed to create fails hence it checks if the thread has been interrupted before it loops over.

  The timeScorer thread is terminated also by roboMove() at the same time. Simply calling interrupt will allow it to gracefully terminate.

In summary, the Map class uses two threads (timeScorer and wallExecute) to manage game logic, update the UI, and coordinate the creation of walls. Proper synchronisation is used to handle shared resources, and threads are terminated gracefully when the game ends

---

Scalability

The main issue my design would face is performance. I believe my program may experience noticeable lag and reduced responsiveness, especially in a larger version. This performance challenge primarily stems from how the robots are rendered onto the grid.
In my design, all robots are stored in one map, and all walls are stored in another map. The requestLayout() method loops through these maps every time it is called. This approach is necessary because failing to do so would result in the omission of robots or walls that haven't undergone changes. To address this issue, I explored potential solutions, including running the rendering process on a separate thread, drawing to a temporary canvas, and subsequently adding the canvas to the display. These optimisations aim to improve the program's performance and ensure smoother execution, particularly when handling a larger-scale version of the game.

One notable issue in the current program design is the limitation imposed by running each robot on a dedicated thread. While this approach works well for smaller projects, it may become impractical and resource-intensive as the project scales up. To address this concern, a viable trade-off would involve modifying the Robot class's run function. Instead of running independently in an infinite loop, robots could be added to a queue. At the end of the queue, multiple threads from a thread pool would dequeue and execute these robots. This adjustment would allow for the simulation of a large number of robots operating independently without the need for an excessive number of threads. However, it's important to note that this modification might introduce minor delays between each robot's execution, typically in the range of milliseconds and not all robots will be moving at the same time.

The way I have synchronised processes in my design may also impact performance, potentially leading to a bottleneck. This is because, at any given time, only one wall or robot is allowed to edit the map. With a larger grid size or a higher number of objects, this synchronisation approach could significantly slow down the program's execution. Implement more fine grained synchronisation but that can also cause some very convoluted code.

Some other non functional requirements we have to consider is Resource utilisation and portability especially for a game that is quite large.