Assignment 2 (Concurrency)

Party Simulation

Nkosingiphile Gumede

GMDNKO003

1.4) Report

1.4.1) Description of Classes

1. *PartyApp*

The PartyApp class in the main class of the program. It is responsible for initiating all of the program's variables as per the command line parameters. Firstly, the number of threads/ people to be simulated. Secondly, the number of blocks the simulation requires (specified by the width and height variables). This class also maintains the GUI from which the user must be allowed to start the application.

2. *RoomPanel*

The RoomPanel is a JPanel extension which repeatedly paints the components of the board. From the people and grid lines to the entrance, bar, exits. This class is run by a single thread that continuously iterates until the simulation is done. This class need not be changed.

3. *RoomGrid*

The RoomGrid class represents the room. The room is represented by a 2-Dimensional array where each dimension represents a length and breadth respectively. Each position in the room is a unique block which will contain either a person or entry/ exit point. This class does not need to be changed.

4. *GridBlock*

The GridBlock class represents the contents of each and every block position in the room. Each block will either be an entry point, exit point, occupied or not occupied (by a person).

This class is used by the RoomGrid class to assist in ensuring the rooms components are correctly placed and checked. This class also need not be changed.

5. ***CounterDisplay***

The CounterDisplay class is used to store the details of the status bar at the bottom of the user interface. This includes the number of people/ threads inside the room, waiting out the room, and that have left the room. It is initiated via the PartyApp main class and run by thread that repeatedly updates the counter until the simulation is done.

6. ***PeopleCounter***

The PeopleCounter class keeps tracks of the status of the threads/ people. A thread is either inside, outside or has left the party. This class is updated via calls from the PersonMover class. This class needs no changes.

7. ***Person***

The Person class represents a person objects and its associated attributes. This includes their colour, position (in the form of x and y coordinates), speed, and whether or not they are thirsty. The class also needs not be changed.

8. ***PersonMover***

The PersonMover class controls the movements of the people/ threads. Each person is represented by a unique thread run by clicking the start button of the user interface. When a person is thirsty (after entering the party) they will head to the bar for a refreshments. Various other criteria (including randomness) will they determine the direction in which each person heads. This is the class from which various concurrency issues are likely to emerge.

1.4.2) Concurrency Requirements

Basic rules for the simulation

1. No guests must arrive before the start of the party.
2. After the start of the party, people arrive at random times.
3. Guest enter only through the entrance door.
4. After arrival, guests must wait until the entrance door is free (unoccupied) before entering.
5. After entering the venue, guests must occupy a grid square, but no more than one square.
6. People may not share grid squares (at most one person in each grid block).
7. A person may not move into a grid square that is currently occupied.
8. People can move to any unoccupied grid square in the room, but must move one step at a time – they are not allowed to skip squares.
9. Guest may only leave through an exit door.
10. Guests do no leave until after they have had a beverage.
11. The counters must keep accurate track of the people waiting to enter the venue, those inside, and those who have left.
12. The pause button (not currently working) must pause/resume the simulation correctly on successive presses.

Additional rules

13. [*Once you have done the above*] Limit the number people allowed into the room at one time and make people wait to enter if the room is full.
14. People should enter in the order in which they arrived.

Changes Made:

Due to multithreading, this program is likely to face a number of concurrency issues. Mutual exclusion and deadlock prevention are required to ensure thread safety. The usage of atomic variables, method synchronization and barriers ensure that my program is thread safe. Below are specific examples of the concurrency issues I fixed when making the program and (in some case) the associated unique methods I used to overcome these issues:

- Added a call to personArrived method at the beginning of the run method of the PersonMover class. This initiates the waiting variable to the number of people outside with each person object created as a means to fix the score keeper.

- Added a pause method within the PersonMover class which used synchronization to check whether or not the pause variable is set the true. Is this pause variable is indeed true, the thread will sleep within a while loop.

- Added an isFull method which checks whether or not the room is full (the number of people inside the room is equal to greater than the room capacity). If it is, the threads will wait until the room is no longer at full capacity.

- Synchronized moveToBlock() within Person class to ensure that people don't clash by checking surrounding blocks and moving sequentially.

- All getter and setter methods have been synchronized to prevent against the occurrence of race conditions.

- Use of Atomic Boolean for occupied variable of the GridBlock class. This is used as a means to prevent against a race condition, only one thread can access this method at a given time.

- Added an else statement to the run method of the PersonMover class to ensure that a person waits until the first block is empty before entering.