Grocery Shopping in the Age of COVID-19

The project must be done individually with no outside source including Internet - no exceptions.
The objective of this project is to familiarize you with the creation and execution of threads using the Thread class methods. You are permitted to use, when necessary, the following methods to synchronize all threads: **run(), start(), currentThread(), getName(), join(), yield(), sleep(time), isAlive(), getPriority(), setPriority(), interrupt(), isInterrupted()**.

The use of semaphores to synchronize threads is strictly DISALLOWED. Additionally, you are NOT PERMITTED to use any of the following: wait(), notify(), notifyAll(), stop(), the synchronized keyword (for methods or blocks), and any synchronized collections or synchronization tools that were not discussed in class.

You CAN, however, use the modifier **volatile** and the **AtomicInteger** and **AtomicBoolean** classes if you choose to.

Directions: Synchronize the customer, employee, and manager threads in the context of the problem described below. The number of customers should be entered as a command line argument. Please refer to and read carefully the project notes and guidelines before starting the project.

****
As unnerving as it may be, grocery shopping during a pandemic must still be done. Fortunately, management at HalfFoods Supermarket has sorted out how to make the shopping experience at their store as safe as possible for all customers and employees.

First, management has imposed a strict limit on the number of people that can physically be in the store at any given time. As the first customers of the day arrive the store (**use sleep(random_time)** to simulate arrival), they wait outside in line for the store to open **(use Busy Waiting and ensure FCFS among customers)**. The manager of the store opens the store not long after seeing customers starting to queue up outside **(randomly determine when the manager opens the store)**. Once the store's doors open to customers, they are asked to enter in groups to make it possible for them to maintain a distance of 6-feet from one another **(group size is determined by store_capacity).** After the customers in the store have made their purchases and left, the manager will allow the next group of waiting customers into the store.

Inside the store, customers take their time picking up groceries (**use sleep(random_time)**). When done picking up what they need, they rush to the self-checkout so they can quickly pay and leave **(simulate rushing by using getPriority() and setPriority(); increase the priority of the customer, and after a sleep of random time, set its priority back to its default value**). However, about a quarter of the customers in the store are elderly **(randomly choose elderly customers)**, and out of deference to them, all other customers kindly allow them a head start to the checkout line **(non-elderly customers use yield() twice)**. The customers wait in line (**using BW**) until the store employee directs them to a free register. With the best interest of the customer in mind, management asks the customers to use every other self-checkout register **(number of self-checkout registers is determined by numRegister).** When one customer is done paying, the store employee will ask the next customer in line to use one of the free registers **(clearly print a message to the screen that states which customer is sent to which register).**

After all customers in the store leave, the manager sends in the next group of shoppers waiting outside. When the last group of customers enter the store, the manager has finished his duties for the day **(manager thread terminates)**.

As much as every customer would like to leave after purchasing their groceries, the parking lot is absolute mayhem! It's as if the customers have forgotten the rules of the road. Before any of the customers can leave, an accident occurs in the parking lot obstructing the exit! The customers in the first group and every group after that must remain in their cars until help arrives (**have customers sleep for a long time**). Unfortunately for the customers, help arrives just as the last customer of the day is served.

The store employee closes up the store and walks out (**use sleep(random_time)**) to the see disgruntled customers relieved that help has finally arrived **(the employee uses interrupt() to wake up the sleeping customers; <u>make sure you place a message in the catch block of the sleep method so it's clear that the passengers have been interrupted</u>).**

One by one, the shoppers leave and the employee follows along immediately after (**customers leave in ascending order of their ID: Customer-1 leaves, Customer-2 leaves, Customer-3 leaves,…., Employee leaves ; use the isAlive( ) and join( ) methods to accomplish this; make sure the threads print a message to indicate that they've left e.g. Customer-1: leaves the parking lot).**

TO REITERATE, you are not permitted to use monitors, semaphores, collections or any other synchronization tools if they were not discussed in class. You can, however, use the **volatile** modifier and the classes **AtomicInteger** and **AtomicBoolean**. Also make sure that you incorporate in your implementation all the implementation details mentioned above.

A few things to note:

1. Make sure you use ample print statements to describe the actions the threads are taking.
2. Initial values:

   Store_capacity = 6   NumCustomers = 20   numSelf-checkout = 4

Guidelines:

1. Do not submit any code that does not compile and run. If there are parts of the code that contain bugs, <u>comment it out and leave the code in</u>. A program that does not compile nor run will not be graded.

2. Closely follow all the requirements of the project's description.

3. The main method is contained in the main thread. All other thread classes must be manually created by either implementing the Runnable interface or extending the Thread class. **Separate the classes into separate files (do not leave all the classes in one file, create a class for each type of thread). <u>DO NOT create packages</u>.**

4. The project asks that you create different types of threads. This means there is more than one instance of a thread. **No manual specification of each thread's activity is allowed (e.g. no Passenger5.goThroughTheDoor())**

5. **Add the following lines to all the threads you make:**

```
public static long time = System.currentTimeMillis();

public void msg(String m) {
   System.out.println("["+(System.currentTimeMillis()-time)+"] "+getName()+": "+m);
}
```

It is recommended that you initialize the time at the beginning of the main method, so that it is unique to all threads.

6. There should be output messages that describe how the threads are executing. Whenever you want to print something from a thread use: msg("some message about what action is simulated");

7. NAME YOUR THREADS. Here's how the constructors could look like (you may use any variant of this as long as each thread is unique and distinguishable):

```
// Default constructor
public RandomThread(int id) {
    setName("RandomThread-" + id);
}
```

8. Design an OOP program. All thread-related tasks must be specified in their respective classes, no class body should be empty.

10. thread.sleep() is not busy wait.   while (expr) {..} is busy wait.

11. "Synchronized" is not a FCFS implementation. The “synchronized” keyword in Java allows a lock on the method, any thread that accesses the lock first will control that block of code; it is used to enforce mutual exclusion on the critical section. **FCFS should be implemented in a queue or other data structure**.

12. DO NOT USE System.exit(0); the threads are supposed to terminate naturally by running to the end of their run methods.

13. A command line argument must be implemented to allow changes to the **nCustomer** variable.

14. Javadoc is not required. Proper basic commenting explaining the flow of the program, self-explanatory variable names, correct whitespace and indentations are required.

A helpful tip:
-If you run into some synchronization issues, and don't know which thread or threads are causing it, press F11 which will run the program in debug mode. You will clearly see the thread names in the debug perspective.

<u>Setting up project/Submission:</u>
For those that use Eclipse:
Name your project as follows: LASTNAME_FIRSTNAME_CSXXX_PY, where LASTNAME is your last name, FIRSTNAME is your first name, XXX is your course, and Y is the current project number.

For example: Doe_John_CS340_p1

To submit:
-Right click on your project and click export
-Click on General (expand it)
-Select Archive File
-Select your project (make sure that .classpath and .project are also selected)
-Click Browse, select where you want to save it to and name it as LASTNAME_FIRSTNAME_CSXXX_PY
-Select Save in zip format (.zip)
-Press Finish

PLEASE UPLOAD YOUR FILE ON BLACKBOARD ON THE CORRESPONDING COLUMN.

The project must be done individually with no use of other sources including Internet.

No plagiarism, no cheating.