Grocery Shopping in the Age of COVID-19
PROJECT 2

The project must be done individually with no outside sources including Internet - no exceptions.

You are asked to synchronize the threads of the following story using semaphores and operations on semaphores. Do NOT use busy waiting.

The initial regular weight for project 2 (as a programming project) was 6%. You have the following choices:

1. You can submit a pseudo-code implementation; in that case the project will have a weight of 5% instead of 6 % (you lose 1% on the weight).

Due Date: Mon 22

Or

2. You can submit a java-code implementation; in that case the project weight will be 7% instead of 6% (you get 1% EC on project 2's weight). I wish I could give you more but too many of you are plagiarizing.

Due Date: Wed 24 (with no penalty until Thu 25 5:00PM; NO SUBMISSION AFTER 5PM)

DO NOT SUBMIT BOTH TYPES OF IMPLEMENTATION: EITHER YOU SUBMIT THE PSEUDO-CODE, OR YOU SUBMIT THE JAVA-CODE.

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. When the first customers of the day arrive at the store (**use sleep(random_time)** to simulate arrival), 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 – use semaphores for the grouping).** The manager of the store opens the store not long after seeing customers grouping **(randomly determine when the manager opens the store)** and allows the first group to enter. After the customers in the store have made their purchases and leave, 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)**). Next customers **wait** to pay. However, about a quarter of the customers in the store are elderly **(randomly choose elderly customers).** There is a store employee who directs them (use semaphores) to the appropriate register. There are **numRegister** registers, one for the elderly and the others for the rest of the customers (use semaphores). Once at the register, the customer waits to pay (use semaphores). The employees will let the customer know when done (use semaphores). 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! Customers must **wait** until help arrives. Unfortunately for the customers, help arrives just as the last customer of the day is served (**use semaphores).**

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 an array of semaphores to accomplish this; make sure the threads print a message to indicate that they've left e.g. Customer-1: leaves the parking lot)**

The synchronization should be implemented through Java semaphores and operations on semaphores (acquire and release)

For Mutual Exclusion implementation use **Mutex semaphores**, not volatile variables.

For semaphore constructors, use ONLY:
Semaphore(int permits, boolean fair)
Creates a Semaphore with the given number of permits and the given fairness setting.

In methods use ONLY: acquire(), release(); You can also use:

getQueueLength()
Returns an estimate of the number of threads waiting to acquire.

hasQueuedThreads()
Queries whether any threads are waiting to acquire.

DO NOT USE ANY OF THE OTHER METHODS of the semaphore class, besides the ones mentioned above.

Any wait must be implemented using P(semaphores) (acquire).

Any shared variable must be protected by a mutex semaphore such that Mutual Exclusion is implemented.

Document your project and explain the purpose and the initialization of each semaphore.

DO NOT use synchronized methods (beside the operations on semaphores).

Do NOT use wait( ), notify( ) or notifyAll( ) which are monitor methods. Whenever a synchronization issue can be resolved use semaphores and not a different type of implementation.

You should keep the concurrency of the threads as high as possible, however the access to shared structures has to be done in a Mutual Exclusive fashion, using a mutex semaphore.

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 numRegisters = numEmployee = 3

Guidelines:

1. Do not submit any code that does not compile and run. If there are parts of the code that contain bugs, comment it out and leave the code in. 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). DO NOT create packages.**

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. Any wait must be implements using the acquire method.

11. 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.

Setting up project/Submission:

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

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

No plagiarism, no cheating.

1. Pseudo-code implementation

You are asked to synchronize the threads of the story using semaphores and operations on semaphores. Do NOT use busy waiting.
Use pseudo-code similar to the one used in class (NOT java pseudo-code).
Mention the operations that can be simulated by a fixed or random sleep_time.
Your documentation should be clear and extensive.
Explain the reason for each semaphore that you used in your implementation.
Explain each semaphore type and initialization.
Discuss the possible flows of your implementation. Deadlock is not allowed.