Machine Problem 4 (MP4) – Ratsie's Simulation

Due: 10/15/2022 08:59:59 am

Background. Ratsie's was a College Park institution that nurtured generations of Maryland students through the rigors of college life. The food served there was not on anyone's "nutritionally recommended" list, but whenever you needed a dose of sodium and saturated fat, it was hard to beat. It closed in early 2015.

Goal. In this project, you will write a simulation for (a highly simplified) Ratsie's (-like restaurant). The simulation has five parameters: the number of customers wishing to enter Ratsie's; the number of tables in the restaurant; the number of cooks in the kitchen that fill orders; the count of each type of machine in the kitchen used for producing food; and a flag as to whether customers' orders will be randomized.

Ratsie's customers place their orders (lists of food items, one list per customer) when they enter. Available cooks then handle these orders. Each cook handles one order at a time. A cook handles an order by using machines to cook the food items. Each type of food item requires its own kind of machine in order to prepare; each type of machine has a "capacity" indicating how many of its food items can be prepared at the same time. Cooks may also use multiple types of machine at once. For example, suppose that a cook is preparing an order containing fries and a pizza, while another cook is preparing an order of fries. If the fryers have spare "count" of at least two, then both cooks can start cooking their fries. The first cook can also start cooking the pizza at the same time as the fries, assuming the ovens have spare "count" of at least one.

Our version of Ratsie's will have four food items, each made by a specific machine type: an order of fries, made by the Fryers, takes 300s (i.e. 300 seconds) to make; a pizza, made by the Ovens, takes 900s to make; a toasted sub, made by the Grill Presses, takes 500s to make; and a glass of soda, made by the Soda Machines, takes 20s to make. The food information is summarized in the table below.

Machines	Food	Cook Time (s)	Simulated Cook Time (ms)
Fryers	fries	300	30
Ovens	pizza	900	90
Grill Presses	subs	500	50
Soda Machines	soda	20	2

Getting started. The project skeleton can be downloaded as a zip file from the course website.

Language. The autograder is set up with JDK 17. Feel free to use any non-preview Java language features such as local-variable type inference, switch expressions, records and sealed classes.

Restrictions. The focus of this project is to learn how to use synchronizer. ***The only java.util.concurrent classes allowed for this project are:***

- ***java.util.concurrent.Semaphore:***

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/Semaphore.html

- ***java.util.concurrent.CountDownLatch:***

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/CountDownLatch.html

- ***java.util.concurrent.CyclicBarrier:***

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/CyclicBarrier.html

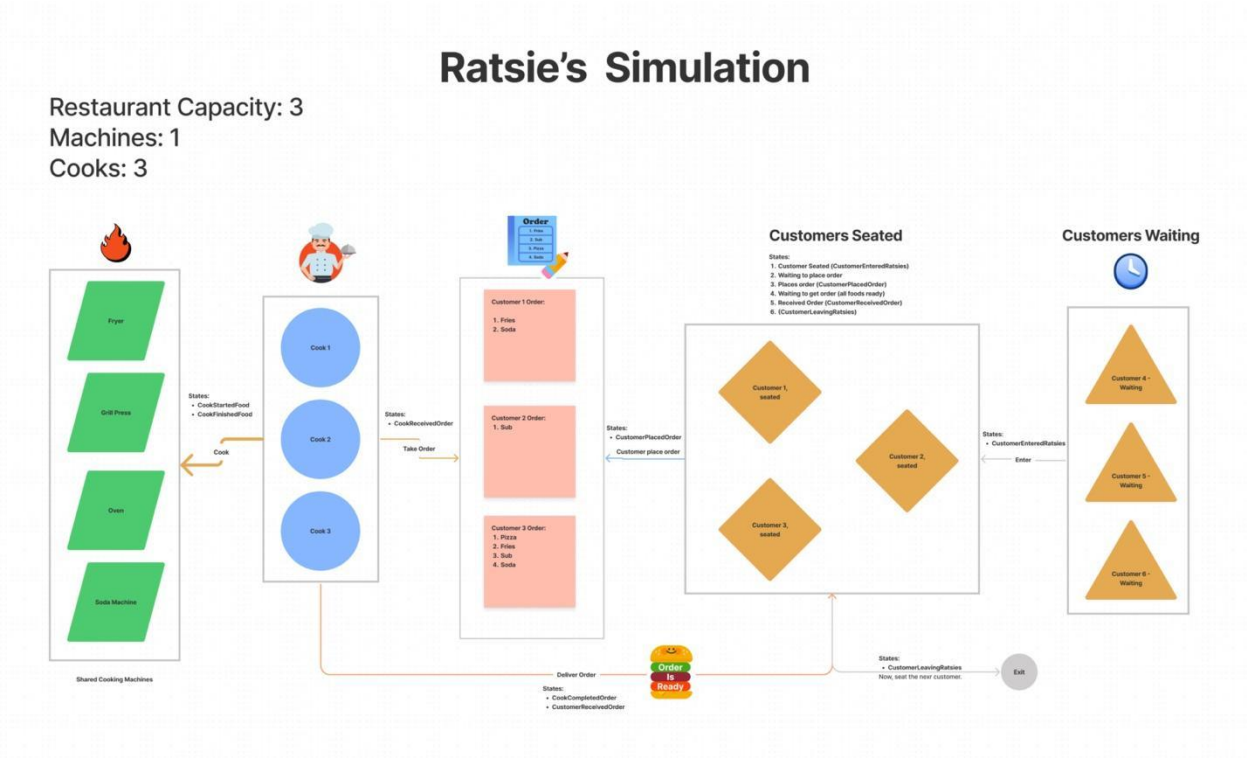
You should familiarize yourself with the following three Java docs to ensure you use each synchronizer properly. You may use the regular (i.e. non-synchronized) version of collections (e.g. HashMaps, ArrayLists) if you wish.

In addition, you must use the following classes and the methods specified for them (most of which you'll be writing). All of these are available in the skeleton code. You may implement any other methods you need for these classes. You may also implement additional private classes or new helper public classes as needed; just make sure they are in the same folder/package as the rest of the project files. Note that files tagged with a ***** in the list below should not be modified (see footnote).

- `Cook.java` – runs in a thread; makes orders provided by customers
- `Customer.java` – runs in a thread to implement a Ratsie's patron
- `Machines.java` – defines class of machines for cooking the food items
- `Simulation.java` – contains `main()` method, which is the simulation entry point for manual testing. Automated testing will call the `runSimulation()` method directly and will not automatically reset static variables, so make sure your program works across consecutive calls to `runSimulation()` (the current version of the main method shows you an example of how we'll call it).
- `Validate.java` – defines method to validate the results of the simulation. You will use this method to check the results of testing your code.

- `Food.java`[1] – contains the definition of food items
- `FoodType.java`* – contains the four food types we have defined above
- `SimulationEvent.java`* – instances represent interesting events that occur during the simulation

Specification in Detail.



- `Simulation`
  - `main(String args[])`: This class is the entry point of the simulation, which is initiated by the main method. While the simulation runs, it will generate events, which are instances of the class `SimulationEvent`. Each event should be printed immediately, via its `toString()` method, and logged for later validation by the `Validate.validateSimulation()` method. We have given you an events list to use to hold these events.

    When the simulation starts, it will generate a `SimulationEvent` via a call to `SimulationEvent.startSimulation()`. The simulation consists of the given number of customers, cooks, and tables. It also involves exactly four types of machine (four `Machines` instances), each with a given "capacity".

1 This class is provided for you and must not be changed in any way.

For this project, there will be two scenarios for orders.

1. If `randomOrders` is set to false, then each Customer places the same order: one portion of fries, one pizza, one sub, and one soda.
2. If `randomOrders` is set to true, then each customer will place an order with a random number of portions of fries, pizzas, subs, and sodas, where each random number will be between 0 and 3, inclusive.

These items will be in a list sent into the `Customer` constructor. Once all Customers have completed, the simulation terminates, shutting down the machines, and calling interrupt on each of the cooks telling them they can go home. Because each of the Runnable `Cook` objects will be running in a thread, a call to `interrupt()` on that thread which will cause the Cook's `run()` method to throw an `InterruptedException`. We have put a try/catch block there for you which invokes `Simulation.logEvent(SimulationEvent.cookEnding(this))` as a result. The last thing the simulation itself will do is generate the event `SimulationEvent.endSimulation()`.

- `Food`
  This class is provided for you. Objects in this class represent kinds of food item. You will create only four kinds food items for your simulation (fries, pizza, sub, and soda, as described above) but your classes should treat food items generically; that is, you should be able to easily change just the Simulation class if you want the simulation to have Customers order different food items or different amounts, without changing any other class.

- `Customer` (implements `Runnable` – needs to run as its own thread)

  - Constructor `Customer(String name, List<Food> order)`: takes the name of the customer and the food list it wishes to order; the format of the name is described below. You may extend this constructor with other parameters if you would find it useful. Each customer's order must be given a unique order number, which is used in generating relevant simulation events; it is easiest to generate this number in the constructor. Assuming the total number of customers is n, all customer names should be of the form "**Customer i**", where i is a number between 0 and n-1, inclusive.

  - `run()`: attempts to enter Ratsie's, places an order, waits for their order, eats the order, leaves. (Note that there can only be as many customers inside the Ratsie's as there are tables at any one time. Other customers must wait for a free table before going into Ratsie's and placing their order.) Customers will generate the following events:
    - Before entering Ratsie's: `SimulationEvent.customerStarting()`
    - After entering Ratsie's: `SimulationEvent.customerEnteredRatsies()`
    - Immediately *before* placing order: `SimulationEvent.customerPlacedOrder()`
    - After receiving order: `SimulationEvent.customerReceivedOrder()`

  - Just before leaving the Ratsie's: `SimulationEvent.customerLeavingRatsies()`

- `Cook` (implements `Runnable` – needs to run as its own thread)

  - Constructor `Cook(String name)`: the parameter is the name of the cook. You may extend this constructor with other parameters if you wish. Assuming the total number of cooks is n, cook names must be unique and be of the form "**Cook i**", where i is a number between 0 and n-1, inclusive.

  - `run()`: waits for orders from the restaurant, then processes the order by submitting each food item to an appropriate machine. Once all machines are finished cooking the desired food, the order is complete, and the Customer is notified. The cook can then go to process the next order. Whenever the cook is interrupted (i.e., some other thread calls the `interrupt()` method on it, which could raise `InterruptedException` if the cook is blocking), then the cook thread should terminates. The cook should not wait for each item to be completed before moving on to the next item. In particular, if the cook needs to cook multiple items, s/he can place all of those requests to the relevant machine(s), one after the other, without waiting for the previous request to be filled. (Note that this applies even if the requests are for the same machine.) For example, one machine could be cooking two batches of fries at the same time for the same order, while another could be making a pizza at the same time for this order as well. Cooks will generate the following events:
    - At startup: `SimulationEvent.cookStarting()`
    - Upon starting an order: `SimulationEvent.cookReceivedOrder()`
    - Upon submitted request to food machine: `SimulationEvent.cookStartedFood()`
    - Upon receiving a completed food item: `SimulationEvent.cookFinishedFood()`
    - Upon completing an order: `SimulationEvent.cookCompletedOrder()`
    - Just before terminating: `SimulationEvent.cookEnding()`

- `Machines` (does NOT implement `Runnable` but could create threads)

  - Constructor `Machines(MachineType machineType, Food food, int capacity)`: the type of the machine, the kind of food it makes, and how many `Food` items may be in-process at once. `MachineType` is an enumerated type that is declared in this class.

  - `makeFood()`: This method is called by a Cook in order to make the Machine's food item. You can extend this method however you like; for example, you can have it take extra parameters or return something other than an `Object`. You will need to implement some means to notify the calling Cook when the food item is finished. Machines will generate the following events:
    - At startup: `SimulationEvent.machinesStarting()`

- When beginning to make a food item: `SimulationEvent.machinesCookingFood()`
- When done making a food item: `SimulationEvent.machinesDoneFood()`
- When shut down, at the end of the simulation: `SimulationEvent.machinesEnding()`

Hint: you will need some way for your `Machines` to cook food items in parallel, up to the counts, but ensure that each item takes the required time. You might do this by having a `Machines` instance spawn threads internally to perform the "work" of cooking the Food. This approach will require some way of communicating a request by a `Cook` to make food to an internal thread, and a way to communicate back to that `Cook` that the food is done. In this simulation, the only thing that will actually be done during the "cooking" time is waiting the proper amount of time (Simulation Time).

Note: you should model the time a food item is actually cooking using a statement of form `Thread.sleep(n)`, where `n` is the simulated cook time of the food. So, modeling the actual cooking of a pizza would be represented via the statement `Thread.sleep(90)`. This means that simulations will run 10,000 times "faster" than real time, since `Thread.sleep(90)` terminates in approximately 600 milliseconds rather than 90 seconds. ***This is a standard practice in simulation development!*** Simulation time and real time are rarely the same. Sometimes simulations run much faster than the phenomena being modeled (as is the case here); other times they run much slower.

You may add helper classes and methods as you see fit to the skeleton code.

Testing. We have allowed a fair amount of freedom in the design of your classes for this project. In particular, we have allowed you to modify the constructors of many classes (such as `Machines` and `Customer`), and pre-written unit tests that access these classes would not work since the tests would not know the signatures of the constructors and methods. In our grading, we will instead use our own simulation-validation routine to check that the simulation output of your code is correct. These checks will be performed multiple times on multiple runs of your code in order to check for things like thread safety.

For these reasons, we have asked you to place all simulation code into a method `Simulation.runSimulation()` which returns a `List` of `SimulationEvent` objects which we can then pass directly to the `Validate.validateSimulation()` method.

Your `validateSimulation()` method will not be graded, although you still must turn one in. (If you do not use the method in your programming, just turn in a method that does nothing. Please note, however, you are strongly advised not to rely only on manually checking of simulation results.) You are encouraged to collaborate with others in this class in developing this method, and it is fine in this case if you and your collaborators turn in the same implementation of it. In this case, please put in a header comment in the `Validate` class

indicating who participated in the development of `validateSimulation()` method you use. ***All other code that you turn in (excepting the files we gave you) must be your own work.***

Sharing of tests (simulation result validation) for this project is encouraged. Public test names correlate to the number of customers cooks, tables, and machine capacity. For example: **test40_8_12_4** means 40 customers, 8 cooks, 12 tables, and 4 foods allowed to cook at once per machine-type. A test ending with "_r" indicates that the Simulation will run with random orders.

One of the easiest ways to fail tests is to have deadlocks in your code. Be sure to test with different combinations of input parameters to ensure that certain scenarios do not lead to deadlock.

Additionally, be wary of initializing static variables outside of the `runSimulation` () method, as they will NOT be reset between consecutive tests on Gradescope. This is expected JUnit behavior, as static variables typically preserve state while JUnit tests assume stateless classes. Improper initialization of static variables can lead to deadlock on Gradescope. You can cover this case locally by calling `runSimulation`() several times within the same JUnit test and checking if your program deadlocks.

As always, it is recommended you start the project early.

Submission. Upload a .zip file containing the `*.java` files inside the `cmsc433` directory to Gradescope. The `*.java` files should be at the base of the .zip file (i.e. the .zip should NOT contain the `cmsc433` directory). Alternatively, drag and drop all the `*.java` files under the `cmsc433` directory into the Gradescope submission window. You may submit an unlimited number of times.

Grading. MP4 is 100% semi-public tests. You will be able to see the score, test output, and test name. You will not have access to the test logic itself.