CS 511: Homework Assignment 2
Due: Sunday 6 October, 11:55pm

1 Assignment Policies

Collaboration Policy. This assignment must be done individually. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming, unless they are members of the same group. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students of different groups. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be reused. Violations will be penalized appropriately.

Late Policy. Late submissions are allowed. The policy is 2 points off for every hour past the deadline.

2 Concepts Covered

- The use of semaphores to synchronize threads
- The use of thread pools to handle large number of threads
- The use of a count down latch to wait for the termination of multiple threads

3 Problem Description

You are the owner of a bakery that specializes in different types of bread. There are three shelves, one for loaves of rye bread, one for loaves of sourdough bread and one for loaves of wonder bread. At the start of the day, your store bakes 20 loaves each of rye, sourdough and wonder bread and places them on their corresponding shelves. Throughout the day customers come in and pick up the

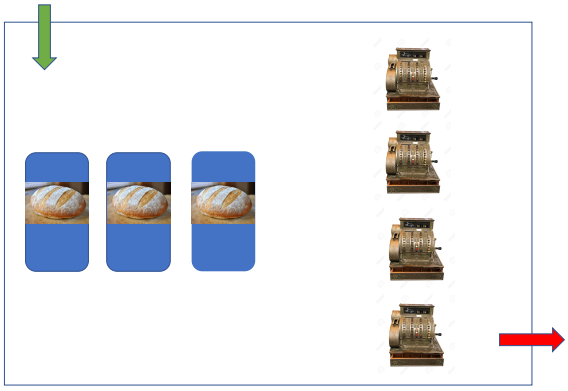

Figure 1: Bakery

loaves of bread from the shelves. They may buy more than one loaf of bread of any type, as described below. Then they make their way to one of the four cashiers that you employ. Those cashiers receive the customer's payments and update the value of the global sales. The customers then leave the store. For safety reasons your store occupancy must be at or below 50 people.

The bakery, as describe above, is illustrated in Figure 1. The green arrow depicts the access door to the bakery and the red arrow the exit door. Only one customer can remove a loaf from a shelf at a time. Only one customer can be at a register at a time.

3.1 The Bakery

The simulation of the bakery starts by spawning one thread executing the code in the method `Bakery.run()`. The file `Assignment2.java` is in charge of doing this. Below we describe the contents of the class `Bakery`. In this particular class, you will have to:

1. Complete the implementation of the method `Bakery.run()`
2. Declare any necessary semaphores

The `Bakery` class has the following fields and methods:

Bakery
private static final int TOTAL_CUSTOMERS = 200 private static final int CAPACITY = 50 private static final int FULL_BREAD = 20 private Map<BreadType, Integer> availableBread private ExecutorService executor private float sales
public void takeBread(BreadType bread) public void addSales(float value) public void run()

The bakery has a store capacity `CAPACITY` and a number of total customers `TOTAL_CUSTOMERS` that visit during the day. This last number will be used as the number of customers in the simulation of the bakery. The bakery has a stock of 20 loaves of each type of bread with a price given in the stub (see `BreadType.java`). Each type of bread is stocked on a different shelf that only one customer can access at a time. The current stock for each type of bread is held in the field `Bakery.availableBread`. The customers will pick between one and three random breads from the shelf. When they take an item from the shelf, it will be taken from the stock. In order for the customer to take from the shelf:

- The shelf must be available (i.e. not in "use" by another customer).
- The bread needs to be in stock.

After they get their bread, customers will proceed to one of the four cashiers and 'buy' the item which adds to the sales variable which is shared between the four cashiers. If all four cashiers are taken, the customer will have to wait their turn. The customer then graciously exits the bakery to let more customers in.

When a bread stock reaches 0 stock, the shelf restocks itself (the method `Bakery.takeBread(BreadType bread)` provided in the stub does this). No one can access the shelf while it is being restocked. The available bread is stored in a ConcurrentHashMap, which is a thread-safe implementation of a HashMap, and assigned to the field `Bakery.availableBread`. We do not want to use the standard HashMap because its "get" and "put" operations are not atomic.

As mentioned above, the `Bakery.takeBread` method (supplied with the stub) takes an item off of the stock and handles the restocking process when necessary. Be sure to acquire the proper semaphore(s) before calling this operation.

3.2 Customer

This section describes the class `Customer`. In this particular class, you are asked to:

1. Implement the constructor `Customer.Customer(Bakery bakery)`.

2. Implement the method `Customer.run()`. Note that each customer thread will be spawned by `Bakery.run()`.

The `Customer` class has the following fields and methods:

Customer

`private Bakery bakery`
`private Random rnd`
`private List<BreadType> shoppingList`
`private int shopTime`
`private int checkoutTime`

`public Customer(Bakery bakery, CountDownLatch doneSignal)`
`public void run()`
`public String toString()`
`private boolean addItem(BreadType bread)`
`private void fillShoppingList()`
`private float getItemsValue()`

A customer has a shopping list called `shoppingList` that determines what bread they take. In addition, they also have a time they spend shopping and a time they spend at the cashier that is randomized. These random values are determined when a Customer is created. The `doneSignal` parameter is explained below.

The operation `fillShoppingList` chooses what items the customers want to buy. The operation `addItem` takes the loaves from the respective shelves and puts it into the customers shopping list (this operation is already implemented for you).

3.3 Solution

Model this scenario using semaphores. These semaphores must allow for a correct use of shared resources. The shared resources in this problem are:

- The shelves: a customer must wait to use the shelf if it is being used by another customer.
- The cashiers: declared in the bakery itself. There are four cashiers available in the bakery that the customers go to check out their items. If the cashiers are busy the customers have to wait until one of the cashiers is free.
- The sales: the sales variable is maintained by all cashiers and is updated when a customer checks out.
- The bakery itself is also a shared resource. However rather than using semaphore for upholding this requirement you are to use the maximum of `CAPACITY` as the size of your thread pool.

Regarding the simulation of the bakery, the bakery should generate customers randomly and have them do their shopping. Customers should have between one and three breads in their shopping list. The shopping itself should take some time but not too much (in the sense that the simulation doesn't slow down too much). Also, the cashiers when checking out should take some time. The simulation should print out events when a customer starts shopping, takes an item from stock, when it buys, and and then when it finishes. Each customer can be uniquely identified using its `hashCode`.

3.3.1 Printing

After all customers have finished shopping, the program should print the total sales from the day before exiting using:

```
System.out.printf("Total sales = %.2fn", sales);
```

The formatting of any other strings printed by your solution are irrelevant. You may choose any formatting.

3.4 List of Classes

Here is a list of the classes you have to define, all included in a package called `Assignment2`

- `Bakery`

  The simulation of the bakery consists in randomly having customers come in to the bakery. You may assume that there are a total of `CAPACITY` that are allowed in the bakery. The `run` method of `Bakery` should be in charge of the simulation itself. The threads corresponding to the customers should be spawned using a thread pool (as explained below).

- `BreadType`

  Is given in the stub and contains information about the bread and prices.

- `Customer`

  Has a shopping list that is randomly generated from the requirements above. Also needs a random value for the time spent shopping and also the time spent at checkout. Contains a method `fillShoppingList` that generates what the customer wants from the shelves.

- `Assignment2` - included in the stub. Starts the program.

```
1 /* start the simulation */
2 public class Assignment2 {
3     public static void main(String[] args) {
4         Thread thread = new Thread(new Bakery());
5         thread.start();
6
```

```
7           try {
8               thread.join();
9           } catch (InterruptedException e) {
10              e.printStackTrace();
11          }
12      }
13  }
```

3.5 A Note on Pool Threads and the Executor Interface

The `Executor` interface[1] or its subinterface `ExecutorService` (which is the one we will be using in this assignment) represents an asynchronous execution mechanism which is capable of executing multiple tasks in the background. Such multiple tasks are referred to as *thread pools*. An `Executor` is normally used instead of explicitly creating threads for managing each of the tasks, i.e., rather than invoking `new Thread(new(RunnableTask())).start()` for each of a set of tasks, you might use:

```
1 Executor executor = anExecutor;
2 executor.execute(new RunnableTask1());
3 executor.execute(new RunnableTask2());
4  ...
```

Thread pools are useful when we need to limit the number of threads running in an application at the same time as there is a performance overhead associated with starting a new thread (each thread is also allocated some memory for its stack, etcetera). Instead of starting a new thread for every task to execute concurrently, the task can be passed to a thread pool. As soon as the pool has any idle threads the task is assigned to one of them and is then executed.

There are many ways of managing thread pools. One easy way is to use the static method `newFixedThreadPool` of the class `Executors` (not to be confused with the `Executor` interface). This method, whose signature is

```
public static ExecutorService newFixedThreadPool(int nThreads)
```

creates a thread pool that reuses a fixed number of threads operating off a shared (unbounded) queue. At any point, at most `n` threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks. The threads in the pool will exist until it is *explicitly* shutdown.

Here is a simple example of `ExecutorService`:

```
1 ExecutorService executorService = Executors.newFixedThreadPool(10);
2
3 executorService.execute(new Runnable() {
```

[1] Found in `java.util.concurrent`

```
    public void run() {
        System.out.println("I am an asynchronous task!");
    }
});

executorService.shutdown();
```

Here is what is happening:

- An `ExecutorService` is created using the `newFixedThreadPool(int)` factory method. This creates a thread pool with 10 threads executing tasks.
- An anonymous class implementation of the `Runnable` interface is passed to the `execute()` method. This causes the `Runnable` to be executed by one of the threads in the `ExecutorService`.
- The `ExecutorService` is shut down, so the all threads finish running. To terminate the threads inside the ExecutorService you call its `shutdown()` method. The `ExecutorService` will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, it shuts down. All tasks submitted to the `ExecutorService` before `shutdown()` is called, are executed.

  If you want to shut down the ExecutorService immediately, you can call the `shutdownNow()` method. This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks. There are no guarantees given about the executing tasks. Perhaps they stop, perhaps the execute until the end. It is a best effort attempt.

3.5.1 Waiting for your threads to finish

After all customers have finished shopping, the program should print the total sales from the day before exiting. In order to wait for all threads in the thread pool to finish, we use a count down latch[2] and issue the shutdown after they have all terminated. The latch itself is declared in the `Bakery` class:

```
public class Bakery implements Runnable {
    ...
    private CountDownLatch doneSignal = new CountDownLatch(TOTAL_CUSTOMERS);
    ...
}
```

We `Bakery.run()` then ends with the following code:

```
 try {
            doneSignal.await();  // wait for all to finish
            System.out.printf("Total sales = %.2f\\n", sales);
```

[2] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CountDownLatch.html

```
4             executor.shutdown();
5         } catch (InterruptedException ie) {
6             ie.printStackTrace();
7         }
```

Each customer thread should receive a reference to `doneSignal` through the constructor `Customer.Customer`. Finally, `Customer.run()` should signal the end the customer's execution with `doneSignal.countDown()`.

4 Submission Instructions

Submit a zip file named `hw2.zip` through Canvas containing the java source files. Important: Please make sure that your assignment compiles before submitting.