CMSC 421: Principles of Operating Systems

Homework 3: GrubDashEats

This homework is due on Tuesday, October 16, at 11:59:59 PM (Eastern daylight time). You *must* use *submit* to turn in your homework like so: `submit cs421_jtang hw3 hw3.c`

Your program must be named `hw3.c`, and it will be compiled on Ubuntu 18.04 as follows:

```
gcc --std=c99 -Wall -O2 -o hw3 hw3.c -pthread -lm
```

(Note the above is *dash*, *dash*, *"std=c99"*, and the other flags likewise are preceded by dashes.) There must not be any compilation warnings in your submission; **warnings will result in grading penalties**. In addition, your code file must be properly indented and have a file header comment, as described on the coding conventions page.

You are a novice software developer for a startup meal delivery company. You will simulate an urban neighborhood of restaurants and customers. You will also simulate a number of drivers, delivering orders from those restaurants to customers. Your goal is to maximize the number of orders delivered during the simulation time.

Part 1: Starting Up

Your C program accepts two single command-line arguments: the name of a data file and the amount of time to simulate. Call this simulation time **T**. Your program will parse the data file to get the locations of restaurants, customers, and the number of drivers. The file then contains customers' orders. If the user does not provide two arguments, or if the data file is unreadable, then display an error message and quit. Otherwise, parse the file as described in Part 3 below.

Your program will spawn these threads:

1. One thread for each of the delivery drivers.
2. One thread for the restaurants in general; it will accept orders from customers and summon drivers.

The general flow for your program will be:

1. Parse the initial data file contents.
2. Initialize all global data values, including all necessary locks.
3. Spawn threads for each driver.
4. Spawn the restaurant thread.
5. Meanwhile, in the main thread, once per second, display the status of the entire simulation.
6. After time **T**, end the simulation. Each driver thread terminates itself. The restaurant thread also terminates itself.
7. The main thread waits for all threads to terminate. It then displays one final status of the simulation.

Part 2: Parse Data File

The data file format is as follows:

- All lines are newline separated; no line will exceed 80 characters in length. Consider using `fgets()` to read each line from the file.
- The first line will contain a positive integer. This is the number of drivers. Call this number **D**.

- The next ten lines give the coordinates of restaurants, as space-separated integers. See Part 3.
- The next ten lines give the coordinates of customers, as space-separated integers. See Part 3.
- All remaining data file lines describe customers' orders. Each line consists of two fields that are space-separated: a restaurant number and a customer number. All fields are integers. See Part 4.

Within your Ubuntu VM, use the `wget` command to fetch the data file `http://www.csee.umbc.edu/~jtang/cs421.f18/homework/hw3/orders1.data`. **The grader will use a different data file with different filename during grading.** You may not make any assumptions about the simulation time or how many orders there will be. It is possible the restaurant thread will not have processed all meal orders in the data file before the simulation ends.

Part 3: Populate Neighborhood

In this neighborhood, there are **10** restaurants and **10** customers. Locations are described by the coordinates **(X, Y)**, where **X** and **Y** are unsigned integers no greater than nine. In this neighborhood, the position (0, 0) is the southwest corner, (9, 0) is the southeast corner, and (9, 9) is the northeast corner.

The ten lines of the data file following **D** correspond with restaurant 0, 1, and so forth up to restaurant number 9. Each line has two numbers, giving that restaurant's coordinates. Multiple restaurants may be at the same location. The ten lines after that give customers' homes. Like restaurants, each line gives a coordinate, and also correspond to customer number. Multiple customers may be at the same location.

After parsing the first 21 lines of the data file, spawn **D** delivery driver threads, assigning that driver a unique identifier. Each driver starts at location (0, 0). Each thread has its counter that tracks the number orders that driver has delivered; initialize that variable to zero.

Drivers take time traveling between restaurants and customers. The travel time to a location is equal to the Manhattan distance multiplied by **250 milliseconds**. Simulate this travel time by sleeping the thread. For example, traveling from (4, 2) to (1, 0) would require 1250 milliseconds. A driver must first drive to the restaurant to pick up an order (see Part 4), and then drive to the customer's home. When the driver arrives at the customer, increment the driver's order counter and print a message.

Note that drivers idle in the neighborhood when awaiting orders. That is, if a driver is not assigned to deliver an order (see Part 4), then that driver waits at his current location. Furthermore, a driver may only deliver **one order at a time**. He may not pick up multiple orders, even if those orders are destined to the same customer.

Part 4: Generate Orders

The restaurant thread is to treat all remaining data file lines as orders. Each line has two fields: a restaurant number, and a customer number.

When the simulation begins, the restaurant thread reads the next data file line. If the restaurant number is non-negative, generate an order. Otherwise, if the restaurant number is negative, make the restaurant thread to sleep for **250 milliseconds** before processing the next line. There may be zero, one, or multiple orders for any given 250 millisecond time slice.

If the restaurant thread runs out of lines to process, then that thread terminates itself, even if the simulation has not ended.

When an order is generated, summon a driver to deliver the order. That driver must first travel to the restaurant to pick up the order, and then travel to the customer's home to drop off that order. It is up to you to choose which driver to handle the order, with the caveat that a driver may only deliver one order at a time. When multiple orders arrive simultaneously, or if all drivers are busy, it is also up to you prioritize which order to handle next. Even if a customer has multiple orders from the same restaurant, the driver **may not "combine"** the orders into a single delivery. (The driver may be *assigned* multiple orders, but he can only process one at a time.)

Part 5: Show Periodic Status

While your driver threads are busy traveling around the virtual neighborhood and the restaurant thread is generating orders, the main thread continues running. Once per second, up to **T** seconds, display these values:

- Status of each driver, either idling or actively delivering.
- Number of orders delivered by each driver.
- Sum of all orders successfully delivered by all drivers.

After **T** seconds elapsed, signal to every driver thread to terminate. (How to signal is up to you.) Those driver threads are permitted to complete their currently order; if the driver is idle then the thread is to terminate immediately. Also signal the restaurant thread; that thread ends itself as soon as possible.

After all threads have terminated, display the simulation status once more.

Part 6: Add Required Documentation

In your code, add a comment block that answers the following:

1. In this assignment, which thread(s) are producer(s)? Which thread(s) are consumer(s)? What data item(s) are being produced?
2. While the main thread is displaying the current state of all driver threads, there is [at least one] potential race condition. Describe a scenario that could trigger that race condition.

Sample Output

Here is a sample output from running the program against the above sample data file for four seconds. This program has added extra debugging output, indicated in *italics*, to display what each thread is doing.

```
$ ./hw3 orders1.data 4
Restaurant 0 is at 2, 6
Restaurant 1 is at 8, 9
Restaurant 2 is at 3, 3
Restaurant 3 is at 3, 4
Restaurant 4 is at 0, 3
Restaurant 5 is at 2, 3
Restaurant 6 is at 1, 4
Restaurant 7 is at 5, 1
Restaurant 8 is at 1, 7
Restaurant 9 is at 0, 8
Customer 0 is at 6, 2
Customer 1 is at 8, 8
Customer 2 is at 1, 4
Customer 3 is at 5, 0
Customer 4 is at 9, 8
Customer 5 is at 0, 4
Customer 6 is at 9, 6
Customer 7 is at 4, 9
Customer 8 is at 2, 0
Customer 9 is at 4, 5
After 0 seconds:
  Driver 0: idle, completed 0 orders
  Driver 1: idle, completed 0 orders
  Driver 2: idle, completed 0 orders
  Driver 3: idle, completed 0 orders
  Driver 4: idle, completed 0 orders
  Total completed deliveries: 0
```

```
    ** Restaurant: Assigning order 0 (3 -> 6) to driver 4
    * Driver 4: starting order 0, heading to restaurant 3 (travel time = 1750 ms)
    ** Restaurant: Assigning order 1 (9 -> 1) to driver 3
    * Driver 3: starting order 1, heading to restaurant 9 (travel time = 2000 ms)
    ** Restaurant: Assigning order 2 (2 -> 4) to driver 2
    * Driver 2: starting order 2, heading to restaurant 2 (travel time = 1500 ms)
After 1 seconds:
  Driver 0: idle, completed 0 orders
  Driver 1: idle, completed 0 orders
  Driver 2: busy, completed 0 orders
  Driver 3: busy, completed 0 orders
  Driver 4: busy, completed 0 orders
  Total completed deliveries: 0
    ** Restaurant: Assigning order 3 (5 -> 9) to driver 1
    * Driver 1: starting order 3, heading to restaurant 5 (travel time = 1250 ms)
    ** Restaurant: Assigning order 4 (6 -> 9) to driver 0
    * Driver 0: starting order 4, heading to restaurant 6 (travel time = 1250 ms)
    ** Restaurant: Assigning order 5 (0 -> 7) to driver 0
    ** Restaurant: Assigning order 6 (9 -> 5) to driver 1
    ** Restaurant: Assigning order 7 (3 -> 7) to driver 2
After 2 seconds:
  Driver 0: busy, completed 0 orders
  Driver 1: busy, completed 0 orders
  Driver 2: busy, completed 0 orders
  Driver 3: busy, completed 0 orders
  Driver 4: busy, completed 0 orders
  Total completed deliveries: 0
    * Driver 4: picked up order 0, heading to customer 6 (travel time = 2000 ms)
    ** Restaurant: Assigning order 8 (1 -> 0) to driver 3
    * Driver 2: picked up order 2, heading to customer 4 (travel time = 2750 ms)
    * Driver 1: picked up order 3, heading to customer 9 (travel time = 1000 ms)
    * Driver 0: picked up order 4, heading to customer 9 (travel time = 1000 ms)
    * Driver 3: picked up order 1, heading to customer 1 (travel time = 2000 ms)
After 3 seconds:
  Driver 0: busy, completed 0 orders
  Driver 1: busy, completed 0 orders
  Driver 2: busy, completed 0 orders
  Driver 3: busy, completed 0 orders
  Driver 4: busy, completed 0 orders
  Total completed deliveries: 0
Driver 1: finished order 3
    * Driver 1: starting order 6, heading to restaurant 9 (travel time = 1750 ms)
Driver 0: finished order 4
    * Driver 0: starting order 5, heading to restaurant 0 (travel time = 750 ms)
Driver 4: finished order 0
    * Driver 0: picked up order 5, heading to customer 7 (travel time = 1250 ms)
Driver 3: finished order 1
Driver 2: finished order 2
    * Driver 1: picked up order 6, heading to customer 5 (travel time = 1000 ms)
Driver 0: finished order 5
Driver 1: finished order 6
After 4 seconds:
  Driver 0: idle, completed 2 orders
  Driver 1: idle, completed 2 orders
  Driver 2: idle, completed 1 orders
  Driver 3: idle, completed 1 orders
  Driver 4: idle, completed 1 orders
  Total completed deliveries: 7
```

Other Hints and Notes

- Ask plenty of questions on the Blackboard discussion board.
- At the top of your submission, list any help you received as well as web pages you consulted. Please do not use any URL shorteners, such as goo.gl or TinyURL. Also, do not cite shared data services, such as Pastebin, Dropbox, or Google Drive.
- Assume the data file will correctly parse.

- Use the `usleep()` function to make a thread sleep. Note that this function sleeps in increments of microseconds, so make sure you multiply the sleep time by 1000, to convert from milliseconds to microseconds.
- There are at least two ways you can solve this assignment. The first approach, as demonstrated in the above Sample Output, is for the restaurant thread to explicitly assign orders to individual driver threads. The other way is for the restaurant thread to push orders to a common pool; idle driver threads would then take an available order from that pool. For both designs, the entire program can be written with only a single lock.

Extra Credit

In the sample output above, the restaurant thread assigns an order to the nearest idle driver. If all drivers are busy, the restaurant thread assigns the next order to a random driver. This algorithm can be greatly optimized.

As a friendly competition, and for fame and glory, you may optimize your algorithm. You can earn an additional 10% on this assignment if your drivers can consistently complete more deliveries than the instructor's implementation for a different (and larger) data file. The astute reader will recognize this as the NP-hard vehicle routing problem.

All extra credit submissions will be tested on the same virtual machine. This virtual machine will have more than one CPU allocated to it. Implementations may not cheat, including but not limited to:

- Spawning more than **D** threads.
- Not sleeping enough when traveling between locations.
- Combining deliveries.
- Allowing idle drivers to travel.

The instructor is the final arbitrator of what is considered cheating or not.

If you choose to perform this extra credit, put a comment at the top of your file, alerting the grader.