Burrito Brothers Design Document.

Author: Eric Webb.

Class: CISC 0640 Operating Systems.

University: Nova Southeastern University.

Professor: Greg Simco, Ph.D.

Due: November 4th, 2019.

Table of contents.

Introduction Pg. 3

Methodology. Pg. 3 & 4

Design. Pg. 4 & 5

Architectural Design. Pg. 5 & 6

Interface design. Pg. 6

Testing. Pg. 6

Work Cited. Pg. 7

Certificate of Authorship Pg. 8

**Methodology and Design Guidelines**

**Introduction**

This document goes into detail on the design and methodologies of a multi-threaded simulation of a burrito making restaurant. Separate Threads will be represented by servers, while other threads will represent customers. The problem this challenges, is that threads must not cause concurrent modifications exceptions and alter data at the same time. To solve this issue of concurrent threading and synchronization, we will incorporate the Java Semaphore class to represent locking of threads. This will allow the program to output information in the command line interface in a clean chronological format, while also creating a safer code behind the scenes free of threading complications. Functions and limitations of this project will go as follows. “X” number of customers will be created while only “Y” amount of these customers will be allowed to enter the store at any given time. The remaining “Z” number of customers must wait outside the store until the “Y” number of customers is below its threshold, at which point the outside customer will be added to the inside line. Once inside the restaurant, the new customer is added to the ordering line. Every time a customer is added to the ordering line, it is re-arranged by the customer’s order size from smallest to largest. The First server in line will then grab the first customer in line, which is the customer with the smallest order. They will continue to proceed with the order and cook up to max 3 burritos per customer. If the customer’s burrito order size is decremented by 3 and becomes equal to 0 or less, then that customer will be sent to the register line. If the customer’s order size does not become 0 or less, they will be added to the ordering line again, and put in order depending on smallest order size. While the current server is added to the end of the serving line, and the next server grabs the next smallest order guest to serve. The register line can only hold up to 3 customers and if a third customer is added to it, then the current server will cash out that customer and they will leave satisfied. Meanwhile, that current server will be added back to the end of the serving line and the next server in line grabs the next smallest order customer repeating this cycle. Once all outside customers, ordering line customers, and register line customers have all been handled and left satisfied the remaining 3 servers will clock out and the restaurant will close.

**Methodology**

The details of these studies were conducted through professional databases containing peer reviewed articles. The objective of this methodology was to review these articles for approaches to solve the concurrency and synchronization issues that are common throughout coding practices. It was assumed the concepts like race conditions, deadlocks, and starvation would be obstacles to overcome in this application, and was verified in publications involving semaphores (Marmorstein, 2018 Pg. 128). Following that research was allocated for data structures, and queuing systems that should be executed for this project. Followed by the proper placement of semaphores to protect against concurrent data manipulation. Specifically, the concepts of Acquire and Release to lock threads in a First in First Out queue type architecture (Wang, Liu, & Spear, 2014, P. 198).

**Design**

Scope/Overview

The following overview will encompass information about this design and prove as a guideline to recreating this program. Limitations for deploying this design are a solid understanding of OOP principles and data structures will be needed. Concepts like Classes, Data Design, Architecture, and Human interfaces will be discussed in such a manner that the application can be reproduced.

Class and Object Design

This program will have 4 classes, RunABuisness, Customer, Server, and Restaurant.

The RunABuisness.class will have the main method and creates our threads of servers and customers via loops while starting their run method implementations. Both run methods from the servers and customers logically will be abstracted away in the Restaurant class. Where they will be added to their respective ArrayLists.

The Customers class will be a POJO with the customer Id represented as an integers and Order Size an integer as well. The order size is a pseudorandom number that is created on the Customers initialization in its constructor between 1-25 burritos. Setter and Getters are used to reference data, and an individual method for subtracting burrito order size by 3 will be needed. To make the code a little more human readable a floor value was set in the get customer order size method so that if any order out come becomes under 0 burritos, then it will display as 0. (You cannot have negative burritos as your order.)

The Server class is another POJO that has integers representing its server Id number and setter and getters as well. A Boolean will be used to establish if a Server is clocked in or not.

The Restaurant class is where the bulk of the program will take place. It is where most business logic is carried out and contain many variables, data structures, semaphores, and methods for simulating this burrito restaurant.

Data Design

Basic primitive variables of integers will be used for customer and server identification along with a Boolean to see if a server is clocked in or out. These server and customer objects will mostly be stored in ArrayLists throughout the program all though a Map and List are also used in the sorting of the lines. Initially servers are loaded into an ArrayList, after that Customers are loaded into an ArrayList. Validation occurs to add Customers in the first array to an Inside customers ArrayList, while the remaining Customers are added to an Outside line ArrayList. ArrayLists were chosen because they are a great way to perform round robins by removing index 0 and adding to the end index of an array you can create a looping round robin architecture. As stated a Map and List of Customers are also created for rearranging and comparing. This was chosen as our data structure because it was easy to compare values using the comparator method to order the line in descending order. Binary Semaphores are also established for many processes such as showing the current line, handling the register, or serving food. The Binary Semaphore was chosen because it locks a single thread to that method allowing for locks to occur and mitigate overlapping.

**Architectural Design**

Loading Servers

On initial load of the program 3 instances of servers are created and stored in an ArrayList. This logic is locked by a binary semaphore on each server’s creation and released after the new server’s position is displayed. The servers then wait to be called upon to start serving later on downstream.

Loading Customers

After the servers are loaded customers are then created and added to a total customers ArrayList. Logic is defined to empty the first portion of the list into another ArrayList representing the inside customers. The remaining of that Array list is sent to an outside customers ArrayList. The inside customers are added to a method as new customers to be sorted in the inside line. A semaphore is used lock when a new customer is added to the inside line. It will display the new lines contents and unlock after being displayed then it will continue on to add the next customer. Once all total customers are loaded, the first server in line will serve the first customer in line.

Order Line Adding and Sorting.

When the loaded customers have been added to the line they are put in a Map if not absent already. This map then compares the customers burritos order size and reorders the customers into a List of customers. Again, Semaphores are used to verify no other functions like cooking or adding other customers are happening in parallel. If the ordered list is at the restaurants max capacity, then the first server begins serving the first customer. A customer might end up coming back to this line if their order has not been completed, that is why a semaphore is used to block re-entering the line while another thread is running this operation in parallel.

Serving First Customer.

Serving the first customer is called when all the customers have been loaded. The first Server in the server line Array will serve the first customer in the final sorted List of customers. When the first customer is pulled from the line the new updated line is displayed and the server begins cooking 3 burritos for that customer. The show line logic and cooking logic are executed during this operation and a semaphore must be used before to lock multiple threads from performing these tasks. These are released when one of 3 things happen, A customer is sent to the register and waiting, sent to the register and cashed out, or if the customer is added back to the waiting line in order. If one of those 3 operations happen the semaphore can be released so that another customer can get served by the next first server.

Cooking.

In the cooking method the customer’s order size is decremented by 3. If the order is completed the server and customer get sent to the register with the server, if not the customer is sent back and sorted to the line while the server is sent back to the end of the serving array. The cooking semaphore is then released so these operations can happen.

Register line

The register is an ArrayList of 3 Customers, the first 2 customers wait in line while the server that sent them there is sent back to the end of the server’s line and the first server begins serving again. On the 3rd customer entering the register line the server that sent them there, cashes out the first customer in the register line is a First in Fist out Fashion and the customer leaves satisfied. That server is then added back to the end of the server line and the next server begins serving the next customer in the ordered line.

Adding outside customers to inside line.

When a customer leaves the restaurant satisfied it makes the total number of customers in the restaurant one less than its capacity. If the outside line is not empty then the first customer in the outside line is added to the inside line to be sorted and the serve next customer method is executed.

End of inside orders.

When serving next customer and there are no more outside or inside customers then that server will go straight to the register to cash out a waiting customer.

Remaining Register

When there are no more inside or outside customers and the current server is at the register the first customer is cashed out from the first server the next customer will be cashed out by the next server in line until all customers have left satisfied.

Servers Clock out.

When there are no longer any outside, inside, or register line customers then the server will clock out. The next server in line will try to serve a customer but will have the same effect until all servers have clocked out.

Restaurant closed.

When all servers are clocked out the restaurant is closed. System exit.

**Interface Design**

The human computer interaction will be that the user will run this program and operations will be printed out via command prompt. The user will be able to see threads and operations being performed for review. The customer will be able to enter 1-100 customers for the day.

**Testing**

Testing was done to verify user input. Users can not enter a number larger than 100 or less than 1, also it cannot be a character string. Stress testing was also done in comparison to the number of inside customers and how many customers for the day to make sure other issues didn’t arise. A limit of 100 customers was chosen so that a stack over flow would not happen without any additional configuration on the users end.

**Work Cited**

Marmorstein, R. (2015). Teaching semaphores using... semaphores. *Journal of Computing Sciences in Colleges*, *30*(3), 117–125. ISSN: 1937-4771 EISSN: 1937-4763

Wang, C., Liu, Y., & Spear, M. (2014). Spaa 14: proceedings of the 26th Acm Symposium on Parallelism in Algorithms and Architectures: June 23-25, 2014, Prague, Czech Republic. New York, NY: ACM. doi: 10.1145/2612669.2612681

**Certification of Authorship.**



Submitted to: Professor Greg Simco.

Student’s Name: Eric Webb

Date of Assignment: 11/4/2019

Title of Assignment: Design and Programming Project Assignment.

Certification of Authorship: I hereby certify that I am the author of this document and that any assistance I received in its preparation is fully acknowledged and disclosed in the document. I have also cited all sources from which I obtained data, ideas, or words that are copied directly or paraphrased in the document. Sources are properly credited according to accepted standards for professional publications. I also certify that this paper was prepared by me for this course.

Student's Signature: ERIC WEBB