**Mariah Avalos Methodology and Design Assignment 2**

**Introduction**

***General Overview***

This assignment tackles the issues of concurrency and synchronization in a simulation similar to that of the Dining Philosopher's Problem (Dijkstra, 1971). There are several components to solving the problem; where one customer can be ringing up their order at a given time, another customer can be being served (fill or partial fill of an order), standing in line, or in the line to checkout. There are three servers, all of whom must be locked onto one customer at a time. After all, in a logical sense it would be impossible for a server to have access to the cash register and food preparation with equal attention.

***Language***

The solutions in this project and their implementations are completed in Java using the Semaphore class for control of concurrency.

***How To Run***

Navigate into the source folder provided via Blackboard. Inside will be the class and java files for the project, an output sample, and this documentation guide. Simply navigate to the source folder in the CLI and run javac Burrito.java and java Burrito to compile and run the program respectively.

***Expected Outcomes***

This program was designed with the expectation that the customers in line who had the smallest order would be filled first, regardless of their original position in line. It is also expected that the three servers are only able to do one task at a time, and are not switching tasks. The customers should be taking their proper places (in line, at the register, etc), depending on what stage of the burrito order they are in. Ultimately this program should be able to handle large numbers of customers without any failures.

**Methodology**

***Implementation Overview***

In this implementation, we assume that the building can only hold 15 people. Any more and the customer must wait until someone else leaves. There are three servers, and an unlimited amount of people attempting to enter the building. Each server is restricted to one position, and one position only while doing that particular task. Once the task is completed, the server may continue onto another task that is not locked by another server. It is not assumed the store closes or that the servers stop working when no more customers are presented, as it's a 24 hour service. Aside from the problems of handling concurrency in general, other problems include needing to generate random orders, random customer numbers, timeouts, and customer tracking.

***Problems***

***Concurrency***

The obvious issues with concurrency and this problem is how to get multiple servers running with multiple customers while still able to tie a task to a server but loosely interpret the relationship between a server and a customer in a many to many type of structure.

***Random order and customer generation.***

Command line arguments were optional in this assignment, so one of the issues was how to generate random numbers of burrito orders and random numbers of customers for this project.

***Timeouts***

Timeouts make or break the concurrency behavior and the application of the project.

How long to make timeouts, as well as where to place them was problematic.

***Customer Tracking***

While the concurrency is handled in another problem, keeping track locally of the customer's position/the customer's order was a present issue in the project. More specifically, the issue of tracking the customers in line for burritos, purchasing, or at the counter and assigned a server.

***Approaches***

***Concurrency***

Concurrency was handled in this project by the Java Semaphore and Thread classes. The thread for servers was called first, followed by the thread for customers. Once the threads were executed, the program was set to acquire and release resources as needed, including ingredients and the register.

***Random order and customer generation.***

Random order was handled fairly simply, by generating random numbers for the order sizes and customers in the same manner.

***Timeouts***

Timeouts were determined based on the best performances of several test trials. Certain timeouts, like grabbing ingredients performed better if less time was consumed, which is also reasonable for real life application. Whereas as decent amount of time needed to be allocated for customer acquiring by servers, else the same server was likely to retrieve t the next customer without giving the other servers a chance.

***Customer Tracking***

Customer tracking was handled by removing customers from the line when approaching the counter or register, and replacing them after a partial fill. The customers were then sorted from smallest order to largest order, to ensure the quickest moving line possible.

**Design**

***Data Design***

private int currentCustomerCount

- The customer count tallying all of the people in the store. This helps to keep track of the building capacity violations.

protected int currentCustomerInLine, customerNumber

- The customers in line, as well as the customer number as per their order of entry. These help to keep track of the line size in relation to the customers and control the allocation of customers and their positions later on. Customer numbers also help to print to the console for easy tracking.

protected ArrayList<Customer> line

- The line of customers waiting for burritos to be made.

protected ArrayList<Customer> lineForTheRegister

- The line of people waiting to checkout/pay.

private static BurritoBrothersStore burritoBrothers

- The base class (Burrito Brothers) that created an instance of the store. This is used frequently, as many of the classes were broken up to avoid clutter and assist with readability.

private int serverNumber

- The server number assigned to each server, assigned in the server class. Used mostly for console logging and tracking.

private Customer customerAtCounter

- The current customer at the counter being served. This helps to manipulate only the properties of single customer being served by any given server.

private int customerNumber;

- The customer's number assigned in the Customer Class. This helps to console log for tracking as well as maintain that servers do not serve anyone except that which has the same customer number during any given task.

private int orderSize;

- The order size for the customer, created in the Customer Class.

***Class and Object Design***

***Classes***

***Line***

The line class consisted of a single method called payAtRegister, which handles placing the customer in line for the register after aquiring the register semaphore and then releasing the register semaphore once the customer is in the line for the register. The method will print an error if unable to acquire or add.

***Server***

The server class consists of a few methods, including free and run. Free is a method that does most of the timeouts and dealing with semaphores. If the customer receives a partial fill on an order, the free method will partially fill the order, call the customer class to reduce the order size, set timeouts for the cooking to help complicate the concurrency behavior, return the customer back to the line, and release the serving semaphore to free another server to work with another customer. If the order is less than or equal to three in size, the order is filled with the above but with the exception that instead of the customer being placed back in line, the customer is added to the register line and the checkout phase with the server who helped prepare the burritos is initiated.

***Customer***

***BurritoPrep***

***BurritoBrothersStore***

***Burrito***

***Data Structures***

***ArrayList***

***Thread***

***Semaphore***

***Timeouts***

***Interface Design***

***Test Provisions***

**Citations**

Dijkstra, E.W.: Hierarchical ordering of sequential processes. Acta Inf. 1 (1971) 115–138