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. As a technical example, an expected output of three customers, with customer 1 having an order of 4 burritos, customer 2 an order of 2, customer 3 an order of 10, and customer 4 and order of 1, we would expect:

server 1 → customer 1 and begin filling the ordering

customer 2 enters → server 2 begins filling the order

server 1 completes customer 1's order, but there is a burrito that still needs making.

customer 3 enters.

customer 3 enters line.

customer 2 pays server 2 as his order is complete.

customer 1 renters line.

server 1 → customer 1 and begins filling the rest of the order.

sever 3 → customer 2 and begins filling the ordering

customer 2 leaves.

customer 4 enters and enters the line.

customer 1's order is completed, they pay and checkout with server 1.

server 2 → customer 4 and begins filling the order.

In this case, eventually the next customer to be filled would be customer 4 and customer 10 would be the last to complete, as they would keep getting moved to the back until the smaller orders were fulfilled. See the output file to view how this program stacks up to these expectations on a much larger scale.

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

Criteria

The following are needed to show the program runs successfully as an acceptable solution to the concurrency problems in the BurritoBrothers assignment:

- There should not be any issues with resources being acquired by more than one server.
- Customers with the smallest order sizes should be filled first.
- No more than 15 customers are allowed in the store at a time.
- Customers should not be in multiple locations, i.e. only in line or only at the counter.
- The line sizes should accurately represent the customers in the lines.
- No two customers or servers have the same identification numbers.
- Once a customer is gone, they should not come back with ghost burritos.

Data Design

Defintions:

Semaphore: “A semaphore is...[an abstract data type]...commonly represented by an integer that records the difference between the number of V and P operations” (Andrews, 1989)

Concurrency: “...the decomposability property of a program, algorithm, or problem into order-independent or partially-ordered components or units.” (Lamport, 1978)

Discussion:

This implementation to handle issues in concurrency approached the problems using semaphores, or locks that allow multiple threads access. These semaphores allow for customers and servers to individually access the resources in a specified order, as can be seen earlier in the philosopher's dining problem.

Instead of chopsticks requiring that each philosopher place his down when done and require

both hands to be full before eating this implementation is approaching the problem with a full order fulfillment before leaving the store. With the burrito brothers problem, the servers can be seen as a mitigation – the halfway point between the customer and the end goal of a full order- much like one of the many proposed solutions to the philosophers problem. The customers as the philosophers; waiting on the others to be completed before they can complete their own tasks. And the chopsticks, last but not least, are comparable to the burritos in which the customer must obtain all to move forward with a task.

Using these semaphores along with array lists to keep track of the customers coming in and out of the store, as well as their order progress, helps to allocate the correct resources to the correct servers and customers.

Two approaches were used to overcome the concurrency issue directly; the first being to allocate resources to the smallest order size, or that which would take the shortest time to free up the resource. The second being using the mediator, or the server, to properly allocate the resources where needed so that the customer never needed to directly access the resources without having everything needed to move forward with the processing. This further helps to prevent any resources from being accessed when they should not be.

Class and Object Design

Properties

private int currentCustomerCount

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

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. Part of the BurritoBrothersStore class.

protected ArrayList<Customer> line

- The line of customers waiting for burritos to be made. Part of the BurritoBrothersStore class.

protected ArrayList<Customer> lineForTheRegister

- The line of people waiting to checkout/pay. Part of the BurritoBrothersStore class.

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.

protected Semaphore customerEntered = new Semaphore(1), serving = new Semaphore(0),register = new Semaphore(1), registerLine = new Semaphore(1), counter = new Semaphore(1), lineCount = new Semaphore(1), ingredients = new Semaphore(1);

- The semaphores used in the program. Part of the BurritoBrothersStore class.

private int serverNumber

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

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. Housed in the server class.

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.

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

The customer class build the customer object, which consists of a customer number, getters and setters for the number, and order size generator and order size reducer for partial orders and a getter for the order size. The class is also unique because it contains its own comparable method that is used later in the BurritoBrothersStore class to sort the line by the smallest burrito order size without needing to hardcode a sort.

BurritoPrep

The burrito prep class is akin to the line class in that it also has only one function. These two were designed this way to help eliminate scattered acquire and releases across the bulk of the program in the BurritoBrothersStore class. In this class, there is a set timeout after the ingredients are gathered that allows for the queue to build up realistically while the servers are making the burritos. This helps to complicate the concurrency and show the true effectiveness of the program to resolve the issues that arise from it.

BurritoBrothersStore

The burrito brothers store class is the heart of the program. Within this class are the semaphores, lines, array lists, and logic that make up the movement of the customers throughout the program. This class contains a static implementation of itself, which is later utilized by all the classes to change the properties within them (such as the server sending the customer back to the line in the server class, but the line being located in the burritobrothersstore class). This class also contains the methods enterCustomer, lineActions, currentCustomerAtCounter, cookBurritos, pay, checkout, and run. Run is a function that implements runnable, and beings the calling of enterCustomer to the program which

introduces the customers generated in the Burrito class. The currentCustomerAtCounter handles removing the customer from the line to either partial or fully fill the order. This will also increment the line by one to move the next person in place up by one index. Line actions sorts the line and adds the customers to the line; if their order has not been taken yet and they are a new customer, it console logs the order amount. The cookBurritos method calls the BurritoPrep class while the pay method calls the Line class. The checkout method fully handles the checkout, including freeing the register, timeouts for simulated paying for humorous effect in the comments, and removing the customers from the capacity count.

Burrito

The Burrito class is the runner class of this program. It contains the threads for the servers and the customers and starts the program. This is also the class that is called to run the program from the CLI.

Data Structures

ArrayList

ArrayLists were used to keep track of the customers in the line for burritos and the line for checkout/register.

Thread

Threads were utilized for execution of the program, where one was allocated for the servers while the other for customers.

Semaphore

The semaphore class was used throughout the code for concurrency. It acquired and released register, ergisterLine, lineCount, ingredients, customerEntered, and

serving. These semaphores all represent objects that can only be handled by one person at a time, and therefore are a singly allocated resource in need of locking.

Timeouts

Timeouts were important to the performance of the program. Timeouts for cooking and paying were much lower than the timeout used to wait for another customer to come into the store. Some of the timeouts were independent of the semaphores, such as the cooking. Timeouts such as the waiting on customers were interlaced.

Interface Design

The program is run via the command line with no additional inputs using javac Burrito.java to compile and java Burrito to run.

Architectural Design

Concurrency Control

Concurrency control is handled through semaphores and thread of the BurritoBrothersStore and Burrito classes. This helps to acquire and release resources per server as needed and given the current status of their use (i.e. timeouts).

- Server thread is started before the Customer Thread.
- Each resource is acquired and released via semaphores.

Data Structure Maintenance

The lines for the register and for the burrito service area were both implemented using array lists. This helped to create a custom comparable for sorting the array list by smallest order size and ease in removing and adding items.

- Customers are removed from the line list and added back if partial fill of order.

- Customers are remove permanently from the line and added to the register line if the order was filled fully.

- The line is sorted by smallest order size each time a customer is added into it.

Test Provisions

No test cases or suites were written to test the viability of the program. Instead, multiple runs on random amounts of customers and observation of behavior was used.

Citations

Andrews, Gregory R., A method for solving synchronization problems. Science of Computer Programming, 13(1):1–21, December 1989.

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

Lamport, L., "Time clocks and the ordering of events in a distributed system", *Comm. of the ACM*, July 1978.