This project is due on the 16 of October at 23:59.

Restaurant for Gophers

In this project you will implement a simulation of a restaurant where chefs, waiters, and customers interact concurrently. The restaurant has multiple chefs who prepare orders and multiple waiters who serve the customers. Each chef can only handle one order at a time, and waiters must deliver the food as soon as the chef finishes preparing it.

Your task is to design a system where customers, chefs, and waiters work concurrently, with proper synchronization using Go channels.

Entities

- Customer: Represents a customer placing an order. The customer places a food order and waits for the food to be served. Each customer has a maximum amount of time that they are prepared to wait for their order. If the time limit is exceeded, the order should be cancelled and chefs/waiters should be notified to stop working on it.
- Chef: Chefs prepare food orders. A chef takes one order at a time, prepares it for a specified time, and then signals the waiter to serve it.
- Waiter: Waiters are responsible for serving the food to the customer. Once the chef finishes an order, the waiter picks it up and delivers it to the customer.

System Description and Behavior

When a customer places an order, it should be added to some queue abstraction from which chefs take and prepare orders. Once a chef completes an order, the order should be delivered by a waiter to the appropriate customer (i.e., the one who requested the order), which then acknowledges receiving the food before the waiter can proceed to the next order.

Note that customers will only wait for so long. Each customer entity should have some logic that encodes a time limit, after which they leave if their order is not satisfied. Upon such a timeout, the order should be cancelled and chefs/waiters should be notified to stop working on it. If an order was "in flight" (i.e., between being prepared by a chef and delivered by a waiter), it should be dropped once a waiter picks it up for delivery.

Assumptions and Requirements

Assumptions:

- You may assume that each customer places a single order, but the number of customers in the system should vary over time.
- Each order takes some random amount of time to prepare.
- You may assume that customers cancel their orders "very vocally" (i.e., they can talk with chefs or waiters if needed).

Requirements:

- The simulation should involve multiple customers, chefs, and waiters all working concurrently.
- Use Go routines to simulate the actions of customers, chefs, and waiters (each in separate goroutines)
- Use Go channels to communicate between chefs and waiters, as well as between customers and waiters.
- Each chef and waiter should not pick up multiple orders simultaneously.
- Your implementation should collect some form of log or statistics of the events taking place, showing customers placing orders, chefs picking up orders, preparing them and notifying waiters, waiters delivering the orders to customers, etc.

Grading

Projects are to be completed preferably in groups of two (at most three), which should match with the groups of the mini-project. The standard plagiarism rules apply and will be enforced.

The project deadline will be on the 16 of October at 23:59. (before the first lecture of the next module), enforced by Github Classroom. You must turn in your code and a **brief** report (add a PDF to the repository), documenting the various design choices in your work. The report should try to address the questions below.

Your code **must not** use locks or mutexes. All synchronization must be done using channels, and Go's channel-based `select`. Waitgroups may be used if you find them appropriate, but only to ensure adequate termination of the program. Any of the techniques seen in lecture may be used (but they need not all be used).

Your project will receive a better grade according to the following criteria:

- **Correctness:** How/If are the constraints of the system achieved? (Critical!)
- **Component management:** How do the various system components organize themselves? How do they communicate? What is your design for handling customer timeouts?
- **Data management:** How are data structure accesses managed in order to ensure the absence of data races?
- **Overall code quality**
- **Use of tests to validate your code**

The criteria are not listed in any particular order. The main focus will be on correctness and component management.