Programming in Go – Concurrency Assignment

**Restaurant service simulation using goroutines**

In the restaurant industry, a kitchen deals with many orders at once. While the orders usually follow a first in first out system, cooks usually always work on different orders at the same time.

Suppose you are simulating dinner service at [insert favorite restaurant here]. The restaurant offers appetizers, entrees, and desserts. A table will order **X** number of appetizers, **Y** amount of entrees, and **Z** amount of desserts (0 <= X,Y,Z <= 20). For the “graded” portion of the assignment, you can assume that you will always have enough food to serve all customers during dinner service.

Appetizers will take **1** second to cook. Entrees take **5** seconds to cook. Desserts take **3** seconds to cook.

**Part 1: Creating a concurrent kitchen**

The kitchen receives all orders at once through a text file **orders.txt**.

<https://reesep.github.io/files/orders.txt>

Use the following for creating the file or download the file *here*. Each row in the file represent a single order followed by how many (1) appetizers, (2) entrees, and (3) desserts. For example, order 1 has 3 appetizers, 3 entrees, and 3 desserts. Order 2 has 5 appetizers, 5 entrees, and 1 dessert. The kitchen should wait two seconds before starting the next order.

<https://github.com/reesep/reesep.github.io/blob/master/files/restaurant.go>

Using ***restaurant.go*** as a starting point, read in **orders.txt** and spawn a goroutine for each row in the file that will “cook” every item for that order. Cooking the items for a single order can be done in a sequential manner (All appetizers first, followed by all entrees, then all desserts)

The goroutine that you will spawn for each order should be called **cook\_order** which takes in an order number, # of appetizers for that order, # of entrees for that order, # of desserts for that order, and the wait group.

Please see the sample transcript to understand how the output should be formatted <https://youtu.be/hrlQq-SZsdM>.

**Note:** It is fine if the order of the output is slightly different output than mine, but the order of finishing *should* be the same. I care much more about if you successfully used concurrency to solve a problem

**Part 2: Monitoring Dinner service using channels**

You will now implement one more goroutine **monitor\_orders** that is going to receive messages when the restaurant runs out of appetizers, entrees, or desserts. If the number of appetizers reaches 0, then “Out of Appetizers!” should be sent from **cook\_order** to **monitor\_orders** and printed out. This communication will be done via a channel. You will need to add one more parameter to the **cook\_order** goroutine to pass in the channel. **monitor\_orders** will act as a “receiver”.

Hint: It is not required to use select statements here because all the **cook\_order** goroutines will be sending on the same channel.

To see the sample transcript after implementing Part 2, follow the link: <https://youtu.be/468zG6RuSU0>