CSI3131 – Lab4
Deadlock Prevention (Java)

Objectives

To gain experience with Java semaphores and deadlock prevention techniques using a restaurant application. **Read the lab document carefully before starting**.

The restaurant application serves has a menu, each menu item is a recipe that consist of many ingredients. Customers (threads) make an order from the menu and wait till order is ready to takeout. The order is first put in a shared FIFO. The FIFO implements the producer (customer) / consumer (dispatcher) solution we discussed in class. The dispatcher (thread) consumes orders one-by-one and assign the order to one of 10 chefs working in the restaurant. Dispatcher shares a FIFO with each chef, same implementation as the order FIFO. Chefs (thread) get orders from their FIFO and start to acquire the ingredients from the kitchen. Every ingredient has a designated MUTEX to acquire, think of it as one sault container that is shared by all chefs!
Since recipe ingredients are ordered differently, a deadlock is eminent to happen when chefs compete to acquire the ingredients! Your task is to prevent that from happening.

Part1 – Use predefined order to acquire ingredients

Ingredients are defines as enumerated (ENUM) type. So we can naturally use the enumeration order to make the acquisition. If all chefs are using the same order to acquire the ingredients of their recipe, circular loop of hold-and-wait will never happen. In this part you'll modify the order in which chefs make the acquisition to make this happen.

Hint: look at the run method for chefs, you'll notice that chefs get the ingredients in the order specified in the recipe. We can enforce a strict order by adding an external loop for all possible ingredients and iterate the recipe to check if we need the ingredient, if so we make the request otherwise we do nothing. This is a king of brute-force sorting algorithm.

```
For all possible ingredients A do
  For all recipe ingredient B
    If A = B,  acquire A
```

Part2 – Make single order for a recipe

To prevent deadlock we are eliminating the hold-and-wait of the ingredients. In this task, you have to add functionality to the kitchen (monitor). The following functions are part of my solution:

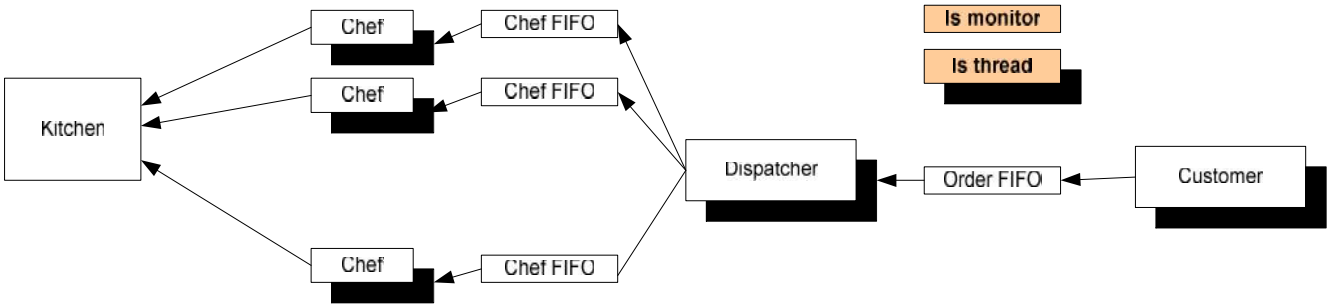
- **`public boolean check_ingredient(Ingredient ingredient)`** check the availability of the ingredient, without actually acquiring it – use **availablePermits** function of semaphore
- **`public boolean check_recipe_ingredients(menu_item recipe)`** use the above function to check that all recipe ingredients are currently available

- **`public void acquire_recipe_ingredients(menu_item recipe, int chef)`**: if the recipe ingredients are available, acquire them and let the chef cook, otherwise block the chef using a mutex and store the recipe for later time
- **`synchronized public void release_recipe_ingredients(menu_item recipe, int chef):`** release all recipe ingredients. Then check all waiting recipes for possible release

The chef can now use the above functions to:

1. get recipe from FIFO
2. acquire recipe ingredients from kitchen
3. cook
4. return ingredients to kitchen
5. go back to 1.

Description



To compile and run the simulation:

- javac *.java
- java `java restaurantApp | tee tmp.txt`
    - `the default configuration is 500 customers, order FIFO size is 50, chef FIFO size is 10`
    - `you can modify that from the command line as follows:`
        - `java restaurantApp 10 5 1 | tee tmp.txt`
        - `10 customers, order FIFO size 5, chef FIFO size 1`

Notes

Notice when you run the simulation there is going to be a deadlock. When you have successfully implemented the solution your simulation will run to completion without any deadlocks.