SMU
SINGAPORE MANAGEMENT UNIVERSITY

Project documentation for

SE301 - Advance Programming & Design

AY 2024-25 – Term 1

Restaurant Ordering System

Image from Adobe Stock

Done by:

Wong En Ting, Kelyn

Lynette Jean Tay

Rachel Yeddu Jaya

1 Context

In the restaurant, there are multiple waiters and chefs. Multiple **waiters** take **orders** continuously, consisting of one or more **dishes**. These dishes of the orders are to be made in the kitchen by one or more available **chefs.** Any chef can take any of the unprepared dishes that are sent to the kitchen by the waiters to make them. Dishes are made out of multiple **ingredients**, which are stored in a shared inventory of the restaurant. This inventory of ingredients can be accessed by one or more chefs to take the ingredients they need to make dishes. When dishes have been made by the chefs, one or more waiters can take the made dish from the kitchen to serve it.

2 The Problem

Our restaurant has only **one kitchen** and **one inventory** storing the ingredients while we have more than one chef and waiter. This means that **multiple** chefs and waiters can access the **same kitchen** with either prepared or unprepared dishes to perform actions on them. These actions can be the waiter placing an unprepared dish request, the waiter serving a prepared dish, or the chef preparing a dish request. **Multiple** chefs can also access the **same inventory of ingredients** to make dishes.

When we represent each waiter and chef as a thread in a multi-threaded program, this becomes an issue as potential race conditions and deadlocks can happen due to the shared kitchen and inventory. The possible race conditions and deadlock and the scenarios that lead to it are as such:

1. **Dish Race Condition**
    - When multiple waiters submit dishes to the shared kitchen simultaneously
    - When multiple waiters try to pick up and serve the same prepared dish simultaneously
    - When multiple chefs attempt to start preparation on the same dish order simultaneously
    - This leads to duplicate submission/preparation/serving of dishes

2. **Ingredient Race Condition**
    - When multiple chefs require and take the same ingredient from the shared inventory of ingredients to prepare different or same dishes simultaneously
    - This leads to ingredient inventory discrepancies

3. **Ingredient Deadlock** (occurs when each ingredient have their own locks)
    - When Chef A needs ingredients X and Y, and has acquired X to make dish 1
    - Chef B needs ingredients Y and X, and has acquired Y to make dish 1
    - Both chefs are now waiting for the other to release the other ingredient they need in order to make the dish, resulting in a deadlock
    - This leads to both chefs not being able to make the dish even though there is the ingredients available to do so

3 Implementation and Solution

The guide to run the application and interpret the results is in the **README.md** file in the source code.

3.1 Implementation

The **RestaurantMain** class is the central controller that orchestrates the restaurant simulation. It starts by loading configuration values from the **Config** class **Singleton** instance where the parameters like the number of chefs, waiters, etc are from the config.txt file. Next, **RestaurantMain** will get the **Singleton** instances of the **Kitchen** and **Inventory** classes. The **Inventory** is populated with the necessary ingredients (e.g., eggs, milk, etc.), which are used by the chefs later during dish preparation. **RestaurantMain** proceeds to create multiple **Waiter** and **Chef** threads concurrently using an executor framework. The **Waiter** threads are responsible for creating orders and submitting the dishes of the order to the **Kitchen** for preparation.

The **Factory Design Pattern** is used by the **Waiter** threads together with the **DishFactory** class. The **DishFactory** creates specific dish objects, such as **SteamedEgg** or **Omelette**, depending on the order. This pattern ensures that dish creation is decoupled from the **Waiter**, allowing for easy addition of new dish types in the future without altering existing code. The **Waiter** then retrieves the created dish from the factory and adds it to the **Kitchen's** toMake list using the addDishToMake() method.

The **Open/Closed Principle (OCP)** is evident in our **AbstractDish** class. This class is open for extension (e.g., new dish types can be added) but closed for modification, ensuring that new functionalities can be added without altering the existing codebase.

The **Liskov Substitution Principle (LSP)** is applied in dish classes, such as **SteamedEgg** and **Omelette**, which extend the **AbstractDish** class. Both dishes can replace any dish type without causing errors in the program's behavior. This principle ensures that derived dish classes can be used in place of their base class without altering the correctness of the program.

Happening concurrently with the **Waiter** threads, each **Chef** thread will retrieve a dish from the toMake list and check the **Inventory** for the necessary ingredients. If the required ingredients are available, the **Chef** uses them by using the useIngredientsForDish() method. After preparation, the dish is marked as "made" using the markDishAsMade() method. If the ingredients are insufficient, the dish is abandoned, and the **Chef** marks it as such using the markDishAsAbandoned() method.

Once a dish has been prepared, the **Waiter** threads retrieve the dish from the **Kitchen** using the getMadeDishToServe() method and serve it. The **Waiter** then marks the dish as "served" using the markDishAsServed() method.

Finally, after all the dishes have been prepared and served, **RestaurantMain** displays the dish history containing all the logs to make dishes, made dishes, and served dishes as well as the current **Inventory** state, showing the remaining ingredients.

3.2 Solution

For our solution, we first identified 2 classes, Kitchen and Inventory, that contain critical sections where **shared resources** are accessed or modified. Thread safety must be ensured to avoid issues like race conditions.

3.2.1 Kitchen

Multiple waiter and chef threads were trying to access and modify the lists (eg., dishesToMake, madeDishes, abandonedDishes, servedDishes, and their count maps) concurrently without proper synchronization. It is necessary to keep track of the count in count maps because only the served list will be filled at the end due to the removal of dishes.

Scenario: Two chefs try to retrieve a dish from dishesToMake list simultaneously. When Chef A retrieves a dish from the list in getDishToMake method, it loses the CPU midway through the execution of return dishesToMake.remove(0); due to context switching by the operating system. Chef B tries to access the same dish, leading to inconsistent or corrupted data where they both end up making the same dish. Similarly, this can happen when waiter and chef threads are trying to update the different lists and dish counts in shared maps (e.g., toMakeDishCountMap and madeDishCountMap), leading to incorrect values if two threads try to increment a count at the same time.

To fix these race conditions, we introduced ReentrantLocks for each shared resource list (toMakeLock, madeLock, servedLock, etc.) to ensure that only one thread could modify these lists at any one time. The locks are ordered and there is no circular dependency, thus there are no deadlocks. We also used a separate lock (dishCountLock) to update shared count maps, ensuring that the critical sections involving dish counts are properly protected. By locking access to shared resources, we prevent threads from interfering with each other, ensuring data consistency.

3.2.2 Inventory

Methods such as addIngredient, useIngredient, and returnIngredient modify the shared stock map that contains the ingredients and their available quantities. Initially, a HashMap is used for the stock which is **not thread-safe**. Without proper synchronization, simultaneous reads and writes to the stock map could lead to inconsistent inventory states.

Our approach to solving potential race conditions in the shared inventory that cause discrepancies is to use ConcurrentHashMap, AtomicInteger, and ReentrantLocks. The stock map was changed to ConcurrentHashMap, which ensures **thread-safe** operations for concurrent reads and writes. This removes the risk of race conditions when accessing and modifying the stock. On top of that, the AtomicInteger for each ingredient's quantity is stored and allows atomic operations like incrementAndGet and decrementAndGet, ensuring that the stock count updates are **thread-safe**.

In the useIngredientsForDish method, locks were acquired for all the ingredients required to prepare a dish. Locks are released in a finally block when all locks are acquired and decremented in the stock. It also ensures that even if an exception occurs, the lock will always be released. However, by doing this there were chances of deadlocks happening because different threads could lock ingredients in

different orders before releasing the lock, creating circular dependencies since multiple ingredients are needed to make a dish.

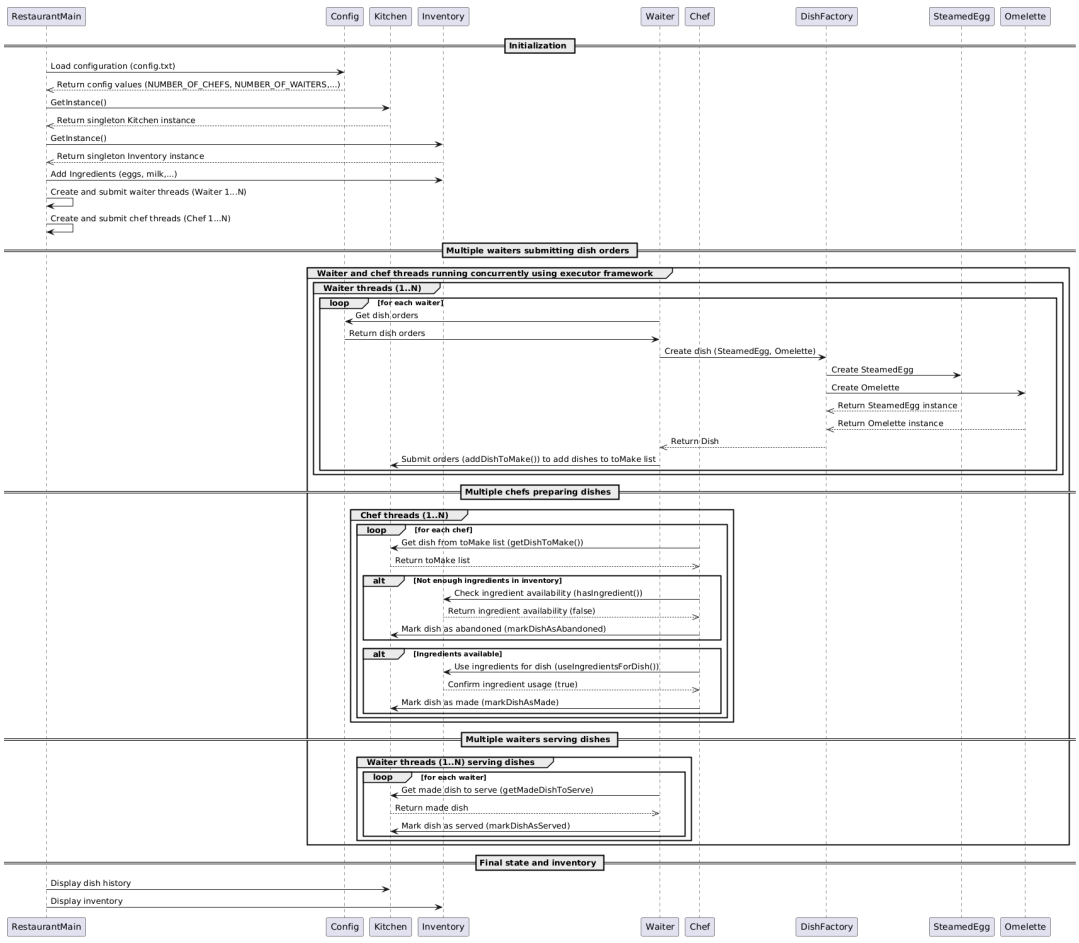
In order to prevent deadlocks, we fixed the order of the lock for each ingredient to be **consistent**. This is done by sorting the ingredient list alphabetically (ingredients.sort(String::compareTo)) before acquiring locks. This ensures that all threads attempt to lock the ingredients in the same order, preventing circular wait conditions and thereby eliminating deadlocks. Additionally, instead of acquiring locks for all ingredients at once, the logic was changed to acquire and use the lock **one ingredient at a time**. This approach further reduces the likelihood of deadlocks by minimizing the duration for which each lock is held.

3.2.3 Why ReentrantLocks?

We chose ReentrantLocks because they offer more flexibility than synchronized blocks, allowing for more fine-grained control over which resources are locked and in what order, thus reducing the likelihood of thread contention and improving the performance of the system in a multithreaded environment. Since we have more write methods than read, we choose to use ReentrantLocks over ReentrantReadWriteLock.

4 UML

4.1 Sequence Diagram



4.2 Class Diagram