SEMAPHORES AND MUTEXES

1 OBJECTIVES

The main goals of this assignment are:

- To learn about semaphores and mutxes as synchronization mechanisms.
- To run several (up to six).threads concurrently.
- To learn to use a queue as a shared buffer.

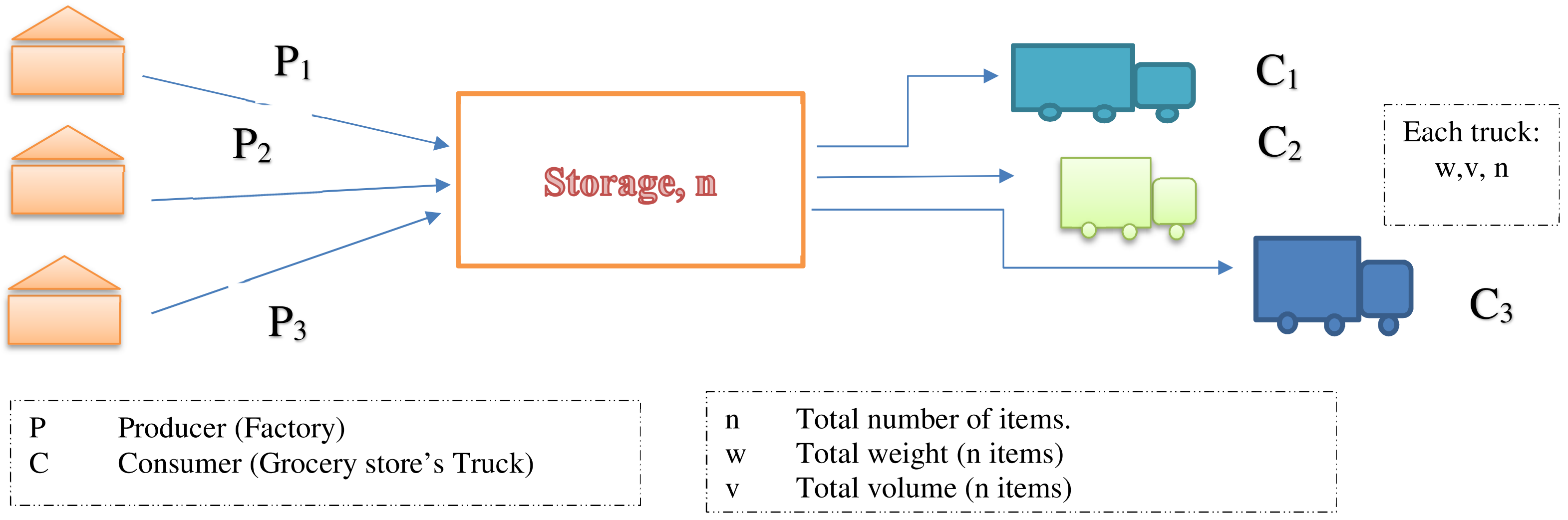
2 DESCRIPTION

You may implement a Producer/Consumer pattern. As before, create a GUI-based application in either C #, or Java (or C++). The GUIs will be provided. For C# (and C++), you may use Visual Studio.

3 FOOD DELIVERY SYSTEM

This assignment simulates a food delivery system. Food items are produced by a number of independent factories (producers), delivered to a common storage, and then transported to a number of grocery stores (consumers) using trucks. The storage is simply a container that has a limited capacity; it cannot take more than a certain total number of items. Even the consumer trucks have limitations. The factories and the trucks can be started/stopped by the buttons. The storage is only responsible for putting and getting items, it has no knowledge of any item data (weight etc).

Working with three producers and three consumers as an example requires that at least 6 threads will be running at the same time, synchronized by semaphores and a mutexes. Also to distinguish easily between the threads in our application, we let each producer be represented by a **Factory** object, each consumer by a **Truck** (or Shop or Store) object and the storage by a **Storage** object.



P	Producer (Factory)
C	Consumer (Grocery store's Truck)

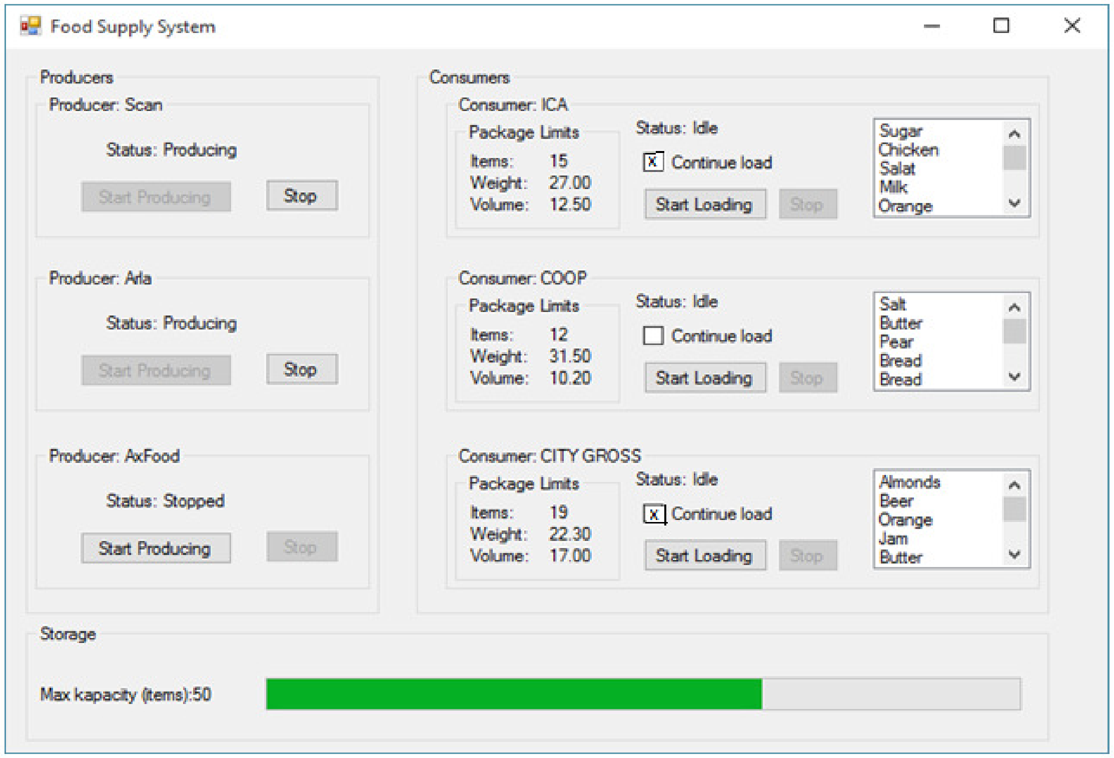
n	Total number of items.
w	Total weight (n items)
v	Total volume (n items)

The Storage object is to maintain a queue structure for the items. The factories produce items and place them in the queue provided the queue is not full. The consumers (trucks) pick up the food from the queue provided the queue is not empty. However, they may not be able to take all the items available from the queue, because they also have capacity limitations. Each truck has a weight limit, a volume limit and a maximum number of items limit, different for the different trucks. The total of these are calculated after each fetch, and as long as all are below limit, yet another can be fetched, but when one of the limits has passed, it stops. The consumer is either done (Continue load is NOT checked) or waits a few seconds, emptying itself and starts over until next limit is passed.

3.1 The GUI

Create a GUI-based application and design the user interface with necessary input/out components. A suggested interface is presented in the figure here.

When a consumer is fully loaded, it either stops the thread or, if the checkbox “Continue load” is checked, it waits for some seconds then it again starts loading from the queue (clearing previous list of items). If the queue is empty the consumer waits before continuing loading, but if the queue is full, the producers wait until the consumers have loaded some items. Try to update the Status labels so they reflect the actual status, like Stopped, Producing or Waiting.



Producer: Scan
Status: Producing

Producer: Arla
Status: Producing

Producer: AxFood
Status: Stopped

Consumer: ICA
Items: 15
Weight: 27.00
Volume: 12.50
Status: Idle
[✓ Continue load]
Sugar
Chicken
Salat
Milk
Orange

Consumer: COOP
Items: 12
Weight: 31.50
Volume: 10.20
Status: Idle
[ ] Continue load
Salt
Butter
Pear
Bread
Bread

Consumer: CITY GROSS
Items: 19
Weight: 22.30
Volume: 17.00
Status: Idle
[✓ Continue load]
Almonds
Beer
Orange
Jam
Butter

Max kapacity (items):50

3.2 The Threads

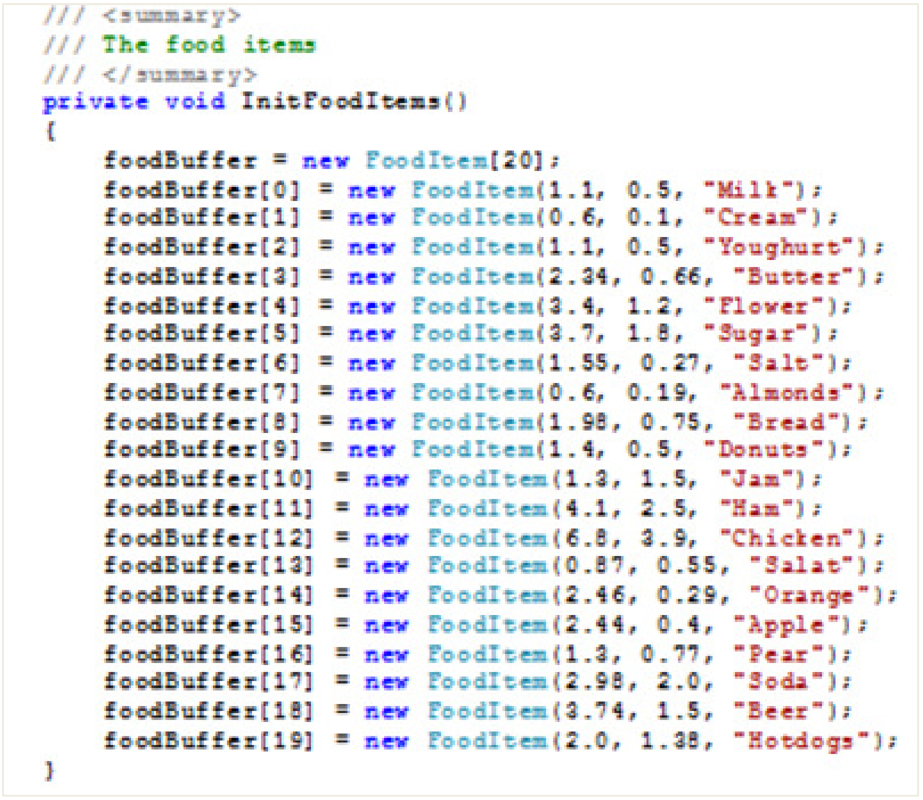
3.2.1 A minimum of **3** producers and **3** consumers are to be used. Write your classes as recommended in above, **Producer** (factory), **FoodItem**, **Consumer** (Truck) and the **Buffer** (storage) using a queue.

3.2.2 The thread synchronization should be made using semaphores, and a mutex for mutual exclusion.

3.2.3 The **FoodItem**-class should have a name, a weight, and a volume field. A collection of food items as in the code snippet can be used as test values.

3.2.4 The **Producer** class should have a buffer and access to the food-item collection. To simulate production of the items, use a randomizer to let each producer create an arbitrary food-item chosen from the collection and place it in the queue. Use also a time interval between each production.

3.2.5 Each Consumer object (truck) takes items from the buffer queue as long as the truck is not full depending on its volume and weight limitation (weight, volume, or number of items). When the truck is full, the consumer is paused some seconds (for instance 5 seconds). After this short pause, if the “Continue loading” button is check, it starts loading again (as if a new truck from the same store is to be loaded).



```
/// <summary>
/// The food items
/// </summary>
private void InitFoodItems()
{
    foodBuffer = new FoodItem[20];
    foodBuffer[0] = new FoodItem(1.1, 0.5, "Milk");
    foodBuffer[1] = new FoodItem(0.6, 0.1, "Cream");
    foodBuffer[2] = new FoodItem(1.1, 0.5, "Youghurt");
    foodBuffer[3] = new FoodItem(2.34, 0.66, "Butter");
    foodBuffer[4] = new FoodItem(3.4, 1.2, "Flower");
    foodBuffer[5] = new FoodItem(3.7, 1.8, "Sugar");
    foodBuffer[6] = new FoodItem(1.55, 0.27, "Salt");
    foodBuffer[7] = new FoodItem(0.6, 0.19, "Almonds");
    foodBuffer[8] = new FoodItem(1.98, 0.75, "Bread");
    foodBuffer[9] = new FoodItem(1.4, 0.5, "Donuts");
    foodBuffer[10] = new FoodItem(1.3, 1.5, "Jam");
    foodBuffer[11] = new FoodItem(4.1, 2.5, "Ham");
    foodBuffer[12] = new FoodItem(6.8, 3.9, "Chicken");
    foodBuffer[13] = new FoodItem(0.87, 0.55, "Salat");
    foodBuffer[14] = new FoodItem(2.46, 0.29, "Orange");
    foodBuffer[15] = new FoodItem(2.44, 0.4, "Apple");
    foodBuffer[16] = new FoodItem(1.3, 0.77, "Pear");
    foodBuffer[17] = new FoodItem(2.98, 2.0, "Soda");
    foodBuffer[18] = new FoodItem(3.74, 1.5, "Beer");
    foodBuffer[19] = new FoodItem(2.0, 1.38, "Hotdogs");
}
```

The sample run image (previous page) shows a situation where all producer threads are running; the consumer threads, ICA and CITY GROSS are also running, but in a stage where their limits of allowable load have reached and are therefore pausing for some seconds, before they start loading again. The COOP thread has only done one load (checkbox unchecked) and stopped when it reached its limits.

4 SPECIFICATIONS AND REQUIREMENTS FOR A PASS GRADE (G)

4.1 To qualify for a pass-grade, you should implement this with good code quality.

4.2 Do your programming work well structured, well organized and always have OOP in mind. Use proper variable and method names, document your code by writing comment in your code.

4.3 NO Logic in GUI! And NO GUI components in logic.

4.4 Comment your code. Use xml headers on classes and methods in C#, use JavaDoc for java files.

4.5 You may certainly bring changes in the application to make it more it more fun full-featured.

4.6 Test your application carefully before submitting.

5 GRADING AND SUBMISSION

Compress all the files, folders and subfolders into a **zip**, **rar, 7z** file, and then upload it via the Assignment page on Canvas. Click the Submit-button "Submit Answer" and attach your file. Do not send your project via mail! Make sure that you submit the correct version of your project and that you have compiled and tested your project before handing in. Be careful not to use any hard-coded file paths (for example path to an image file on your C-drive) in your source code. It will not work on other computers. Projects that do not compile and run correctly, or is done with poor code quality, will be returned for completion and resubmission.

After or before submitting your assignment to the module, show your assignment to your lab leader during the scheduled hours in the labs.

Good Luck!

Farid Naisan,
Course Responsible and Instructor