READERS/WRITERS – 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

This assignment consists of two alternatives. You should implement at least one of them:

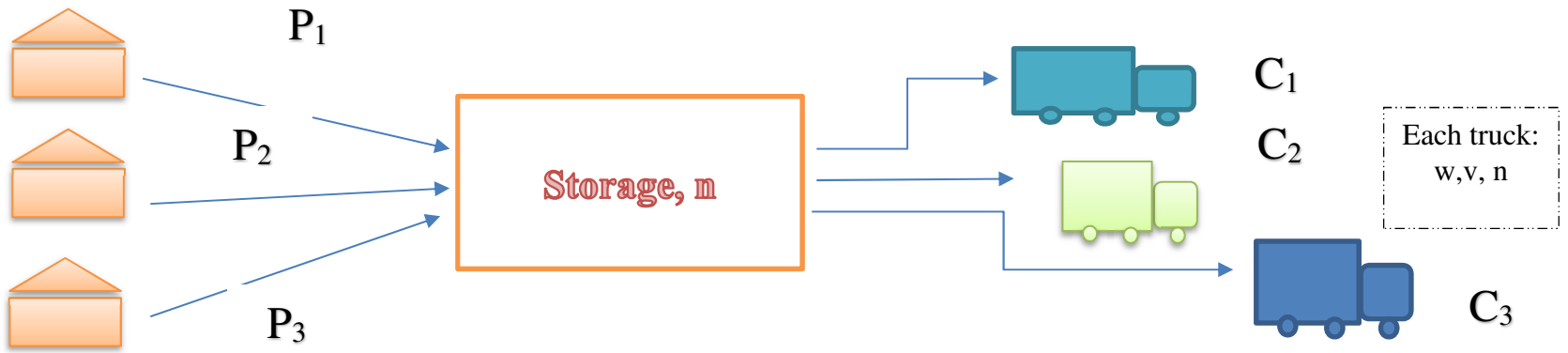
1. Food Supply System.
2. A simple Encryption System.

For the first alternative, you may implement a Producer/Consumer pattern but for the second alternative, a Writer-Reader problem is more suitable. As before, create a GUI-based application either in C# or Java (or C++). The GUIs in Java 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 which

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).



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

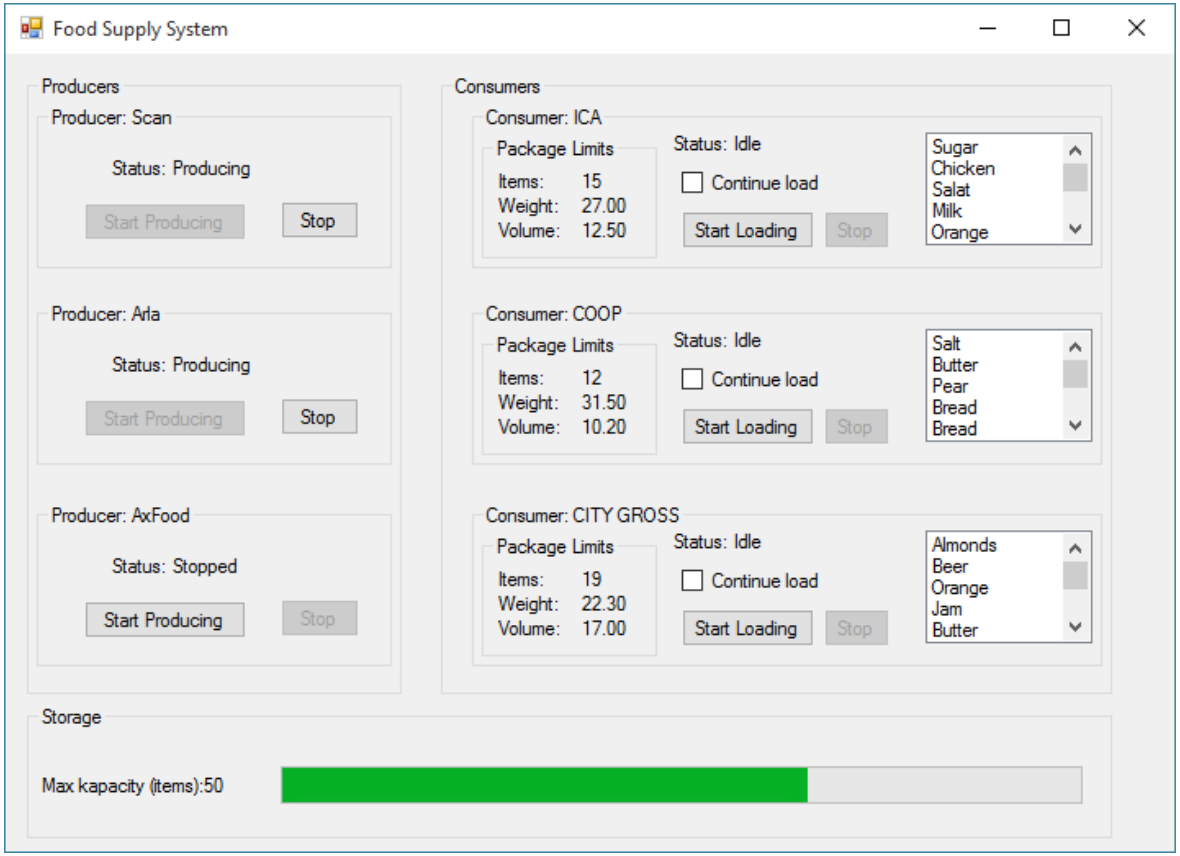
n	Total number of items.
w	Total weight (n items)
v	Total volume (n items)

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.

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 truck. 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.

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).

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.

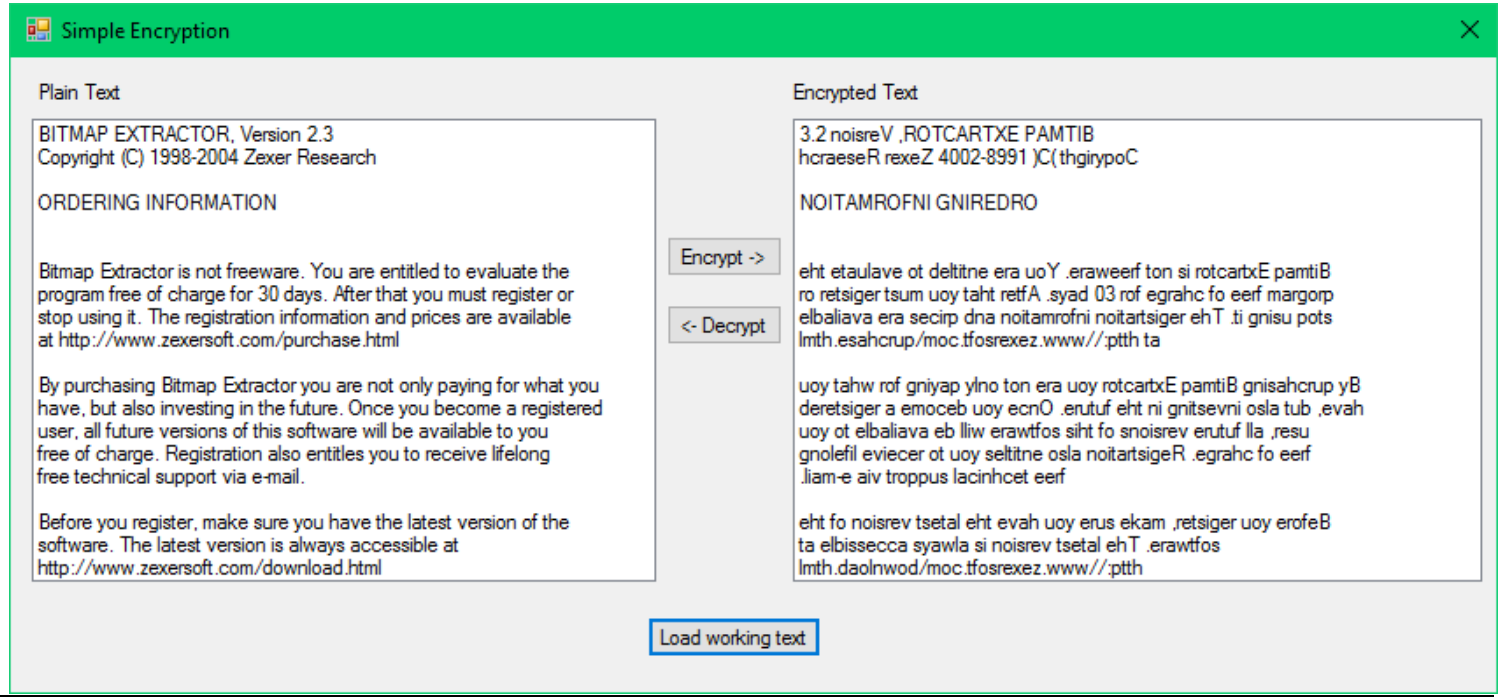
```
/// <summary>
/// The food items
/// </summary>
private void InitFoodItems()
{
    foodBuffer = new FoodItem[20];
    foodBuffer[0] = new FoodItem(1.1, 0.5, "Milk");
    foodBuffer[1] = new FoodItem(0.6, 0.1, "Cream");
    foodBuffer[2] = new FoodItem(1.1, 0.5, "Youghurt");
    foodBuffer[3] = new 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");
}
```

4 SIMPLE ENCRYPTION PROBLEM

Write an application that reads the contents of a text file into a queue of strings (each line is read as a string) and encrypt it by reversing each string, i.e. placing the last character of a string as the first, continuing until the first character is placed as the last one in the reversed string (see the GUI sample below). You may certainly implement any other encrypting/decrypting (or ciphering/deciphering) method, in case you would like to create a more advanced (and interesting) solution.
Start with reading a text file to a string list. This is our working data to send to the writer.

4.1 The GUI

The application should allow the user to select a text file whose content is to be encrypted. In the sample GUI a TextBox control is used to display the text. When the **Encrypt** button is pressed by the user, the text is to be encrypted and displayed in another TextBox object at the right. The process is reversed when the user clicks on the **Decrypt** button. The Encrypt and Decrypt buttons are to be enabled/disabled when necessary.



4.2 The Threads

For the encryption task, you may use three threads:

- **Writer thread**: for writing each string from the text source (the file) to the shared buffer,
- **Encrypter thread**: that reads a string from the buffer and performs the encryption
- **Reader thread**: that reads the encrypted strings and displays the output strings.

For the decryption task, the procedure is reversed although all the three threads are used. The difference is that now the encrypted strings become the input.

4.3 To Do

4.3.1 Create the classes **Writer**, **Encrypter**, **Reader**, and the **Buffer** using queues (Hint: use two queues)

4.3.2 Make sure that the size of the queues are **less** than the number of strings in the source file. Copy the strings to a list of strings, and display them in a plain TextBox.

4.3.3 Let the three threads sleep for a random time between accesses to the shared data in order to slow the process so we can observe the flow of actions. When encryption is done, copy the result strings to a second string list, this is then to be used as input for the decryption operation.

4.3.4 Decryption works in the exact same way, but with the encrypted string list as input to the writer. The output from this operation is then to be displayed as the original text, but the output textbox (at the right side).

4.3.5 The three threads should use semaphores for synchronization and mutex for mutual exclusion.

5 Specifications and Requirements for a pass grade (G)

5.1 To qualify for a pass-grade, you should implement at least one of the above two alternatives with good code quality.

5.2 The application should have a graphical user interface. Design your GUI and the way it should interact with the user. The GUI should show changes to the schedule.

Note: If you are using a “Controller” class, do not send the-reference to UI-class to the Controller methods; send the reference to the control on which your thread is operating.

5.3 Use a **Readers/Writers** model, semaphores and mutexes (binary semaphores in Java) to solve this assignment.

5.4 Test your application carefully before submitting.

6 GRADING AND SUBMISSION

Show your assignment to your lab leader during the scheduled hours in the labs, but before doing so, you must upload your work to Its' L. 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.

Compress all the files, folders and subfolders into a Zip, Rar or 7z file, and then upload it via the Assignment page on It's L. Click the button “Submit Answer” and attach your file. Do not send your project via mail!

Good Luck!

Farid Naisan,
Course Responsible and Instructor