LOGISTIC MANAGEMENT SYSTEM
SEMAPHORES AND MUTEXES

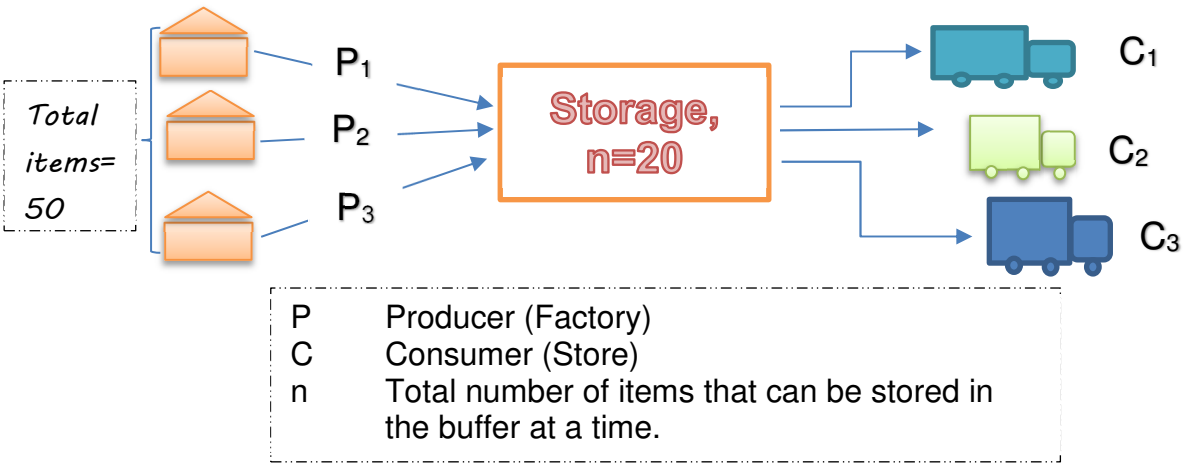
1 OBJECTIVES

The main goals of this assignment are to:

- To gain knowledge about semaphores and mutexes as synchronization mechanisms.
- To run multiple threads concurrently using semaphores and mutes.

2 DESCRIPTION

This assignment simulates a transportation system between product manufacturers and retailer companies. Independent factories (producers) create product items that are then delivered to a common storage. Grocery shops collect the delivered items from the storage. The storage has a limited capacity, which means that there is a maximum number of items that can be stored at any given time. Additionally, we limit the number of producers and consumers to three each as a simplification.



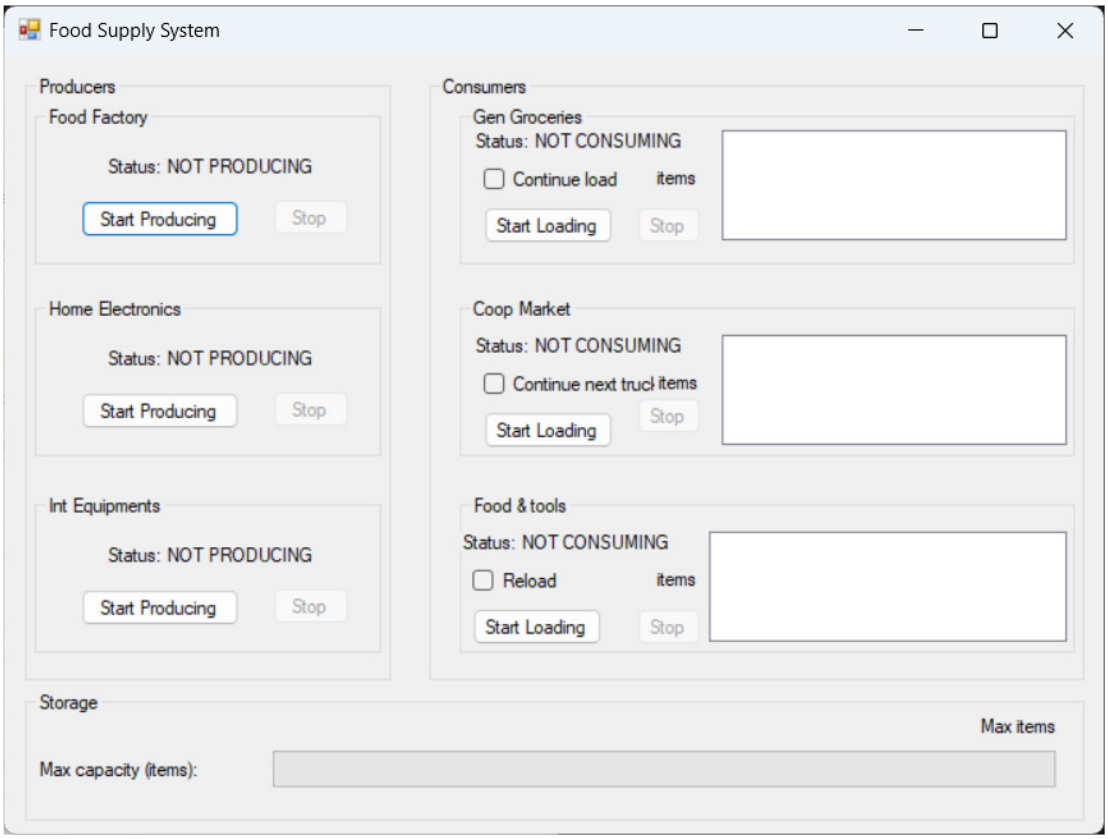
The image above depicts the scenario upon which this assignment is based. It is a simulation of a logistic system that includes producer factories, a shared warehouse, and consumer factories. The producers simulate factories that manufacture items, and their trucks place them in the buffer provided the buffer is not full. Similarly, the consumers (trucks) collect the items from the queue provided the queue is not empty.

The storage object is responsible for coordinating the threads that produce and consume the items. Moreover, the storage object provides a common buffer (placeholders) for maintaining a certain number of items that are stored by the producers and consumed by the consumers. However, the buffer has a limited capacity.

3 REQUIREMENTS:

Develop a multithreaded application that is based on the aforementioned scenario. The application should come with a GUI, making it easy to observe how the threads work concurrently without any interference or race conditions while ensuring safety and liveness properties. To synchronize access to the shared data, meet the requirements by using semaphores and mutexes. To protect the critical section, use a mutex (or binary semaphore in Java). Utilize the semaphores' signaling capabilities appropriately to prevent deadlock, starvation, and other unexpected behavior. For the application, represent each manufacturer with a **Producer** (or **Factory**) object, each retailer with a **Consumer** (or **Store**) object, and the storage with a **Storage** (or **Warehouse**) object.

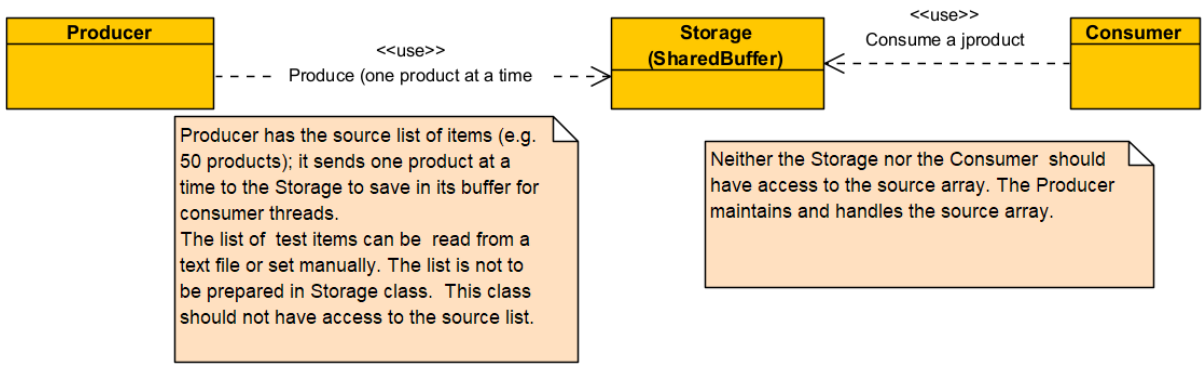
3.1 Below is a suggested GUI design for the application. While the code for C# and Java for this GUI is provided on the assignment page, you are free to design and create your own. You may name the producer and consumer companies in any way you choose.



3.2 The Storage should have a buffer with a limited capacity other than the number of source items. Use a fixed size array or a collection with maximum capacity that is smaller than the number of items to be produced:

```
const/final int buffersize = 20;
Products [ ] buffer = new Product[bufferSize];
```

3.3 Follow the following pattern in designing your solution:



3.4 Once a consumer truck is fully loaded, the user has two options. The user can either stop the thread and restart it or select the option to continue loading. If the checkbox "Continue load" is checked, the consumer truck will wait for a certain amount of time before starting to load items again, clearing the previous list of items.

3.5 If the buffer is empty, the consumer truck will wait until items become available. Conversely, if the buffer is full, the producers will wait until the consumers have loaded some items.

3.6 Update the status labels on the GUI (see the GUI image, presented earlier) for both the producer and consumer threads in a way that reflects the actual status, such as "Not consuming," "Producing," or "Waiting."

Classes and threads:

In addition to GUI and other standard classes such as the start class (Main or Program), it is recommended to create the following: Product, Producer, Consumer, Storage, CategoryType (enum). You may of course name the classes as you desire.

A detailed description of the above is provided in the following paragraphs. A summary is presented at the end of this section.

3.7 The product class

Create a class product with attributes: name, price and a category type:

class Product

- string name
- double price
- Category (Food, Electronic or Tool)

```
enum CategoryType
{
   Food,
   Electrics,
   Tools
 }
```

Implement a **ToString**() method to return a string in the format "name - price" for display on the GUI.

3.8 The Storage class (bounded buffer class)

The Storage class serves as the container class for a shared bounded buffer in the application, and it can use any array or a collection type with a maximum capacity, for example 20 items. The elements of the buffer should represent a product object. The items can be any type of product, such as food items or electronics.

3.9 The class should have methods for producing and consuming, for use by the producer and consumer classes, respectively. Synchronizations should be done in these methods. It is crucial to ensure that the buffer is synchronized using semaphores and mutexes to avoid any interference and race condition.

The threads

3.10 Create a minimum of 3 producer threads and 3 consumer threads, with the option to experiment with more threads if desired.

3.11 Use the classes recommended in previous sections: **Producer** (or **Factory**), **Product**, **Consumer** (or **Store**), and **Storage** (or **SharedBuffer**) using a collection.

3.12 Synchronization of threads should be implemented using semaphores and a mutex for mutual exclusion, with synchronization logic implemented in the **Storage** class.

3.13 Each producer thread has no limitation on the number of items it can produce, but the total number of items produced by all threads cannot exceed a maximum number, for example 50.

3.14 Prepare a test list of items and make available to Produce class. The list can be prepared inside the Producer class as well. It can be copied from an online source, created manually or loaded from a text file.

3.15 The list types that may be used is a fixed-sizes array or collections such as List, Dictionary, HashTable, and LinkedList, as long as they have a maximum capacity that is not too large..

3.16 Each **producer** thread should use a randomizer to pick up an arbitrary item from the collection and place it in the buffer, with a time interval between each production.

3.17 Each consumer should have a limit on the number of items it can load. For example, consumer thread1 can load max 10 items, consumer 2 max15 and consumer 3 max 8 items. A consumer thread can either be stopped and restarted by the user through the GUI, or automatically be reset when the "Continue load" option is selected.

3.18 Each Consumer object (truck) should take items from the buffer list, as long as the truck is not full, depending on its number of items limitation. When the truck is full, the consumer stops loading. If the "Continue load" option is checked, the consumer will start loading again as if a new truck from the same store is being loaded.

Summary

In summary, the assignment requires the creation of a multithreaded application that simulates a transportation system between manufacturers and retailers using three producer threads and three consumer threads. Proper synchronization using semaphores and mutexes should be employed to ensure thread safety and liveness properties.

- The GUI should also display relevant information such as the current status of the threads.
- The Storage class should be implemented using a collection or an array with a maximum capacity, for example **20** items to serve as buffer.
- Each **consumer** thread should have a maximum capacity of items they can carry, e.g., **10, 15,** and **8** for the first, second, and third consumer, respectively.
- There is no maximum limit for each **producer** thread. The total number of items to be produced by all three threads together should be limited to a certain number of items, for example **50**.

4 Submission and General Requirements for a Passing Grade (G)

4.1 To qualify for a pass grade, you should maintain good code quality and a well-organized project. Don't forget to document your code by writing comments.

4.2 You may certainly make changes to the application in order to make it more advanced, and full featured.

4.3 Test your application carefully before submitting; make sure there are no breaking bugs.

Submit the assignment as previously. Upload your solution as a compressed file either before or after showing it to your lab supervisor.

Good Luck!

Farid Naisan,
Course Responsible and Instructor