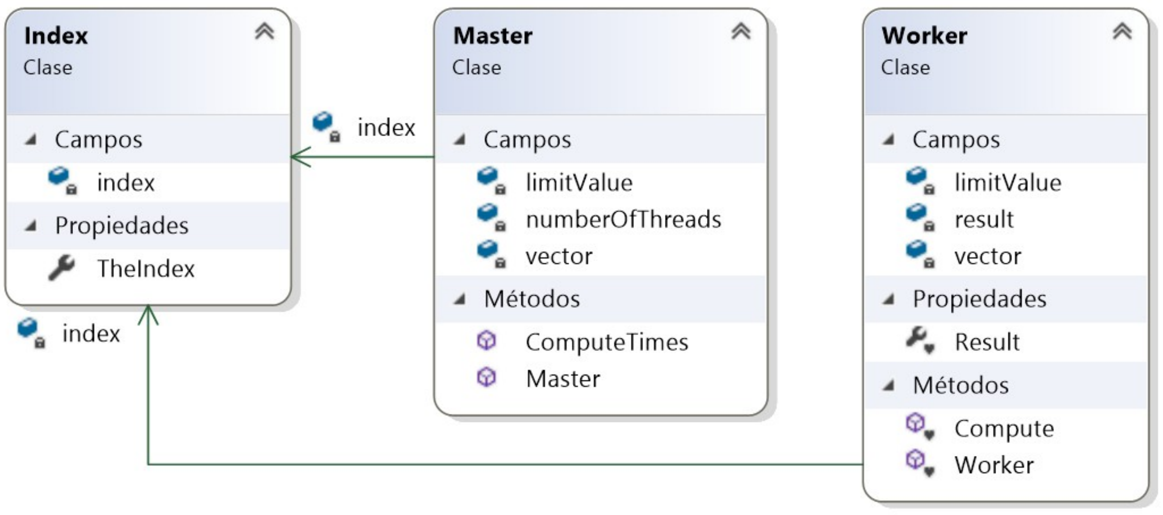
Programming Technologies and Paradigms. Lab 10, presential activity.

Exercise 1. For this exercise, the last mandatory activity (master/worker BitCoin value) will be revisited. Instead of statically partitioning the data, we are going to use C# synchronization mechanisms and therefore no fromIndex and toIntex attributes will be needed. The idea is as follows: we will use a pool of threads and an index to store the position of the BitCoin data vector where the incoming thread will test if the value at that position it is above a threshold (7000 in the example). This index is a shared resource that must be locked when a thread access to it to obtain a copy and increment it. Then it is unlocked and the next thread will do exactly the same. Do not expect performance increasing in this case because of the simple computation involved. This idea can be implemented using two different designs

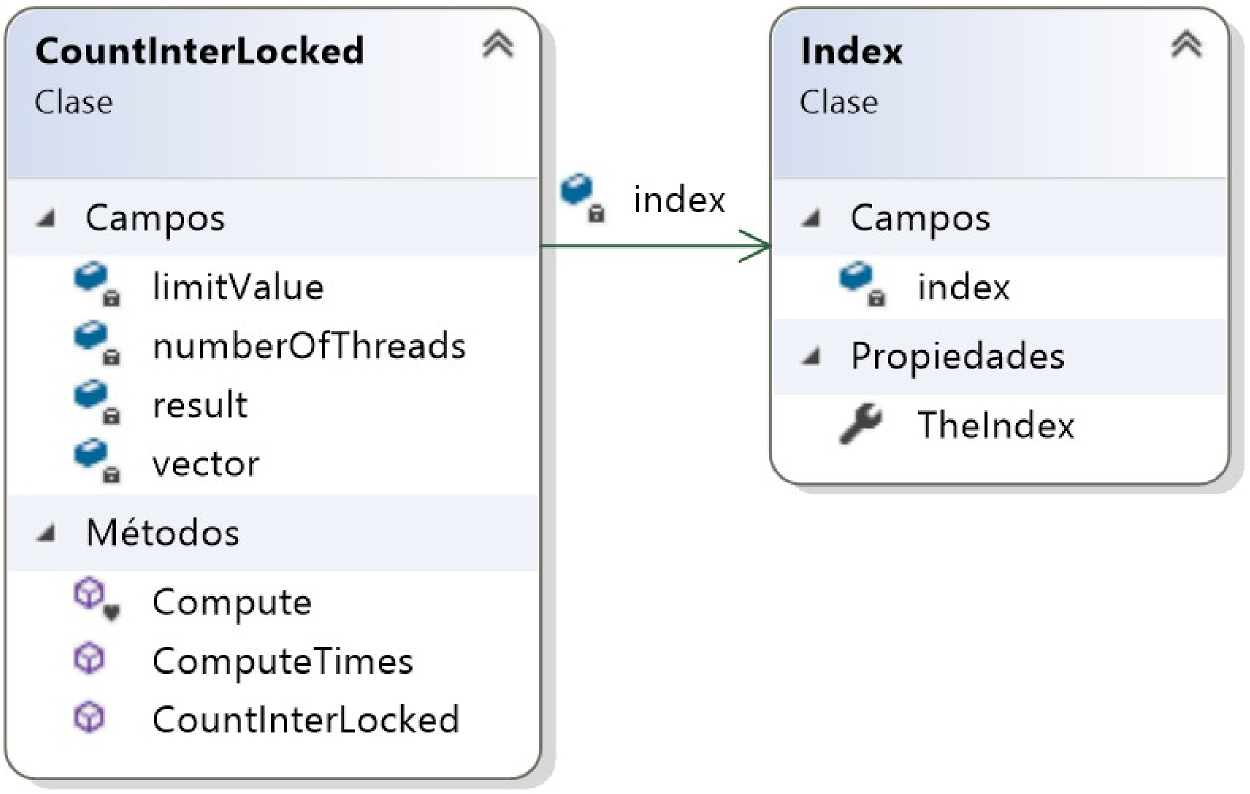
NOTE: I have created the whole set of projects for this lab session, you only have to complete the gaps in the source code. The following is only for illustrating purposes.

1. Create a new solution with the past mandatory activity .cs files. Add the public class Index, with a single int attribute. An instance of this class will represent the mentioned index. Add an Index attribute to the class Master. Master constructor initializes this attribute to zero. Add an Index attribute to the class Worker, modify the constructor adding an Index parameter. Read carefully this: locate the loop where the Worker array is created, create each Worker with a Master Index attribute reference (not the int Index attribute). In this way a reference to the Master attribute is shared among the Workers.



During the execution of each Worker, in each iteration, each Worker makes a copy of the Index int attribute and increments it. Because it is a shared resource, use `lock` to guarantee that no other thread is doing the same concurrently. The copy of the int attribute is where the thread checks in the vector data if the value is above a threshold. If true, increments its result attribute value. Each thread iterates while there are BitCoin values left to check.

2. A second idea is to add a long attribute to the class Master (`result` in the class diagram, will hold the number of times the bitcoin value is above the threshold) and use `Interlocked.Increment` with that attribute from each thread, instead of a decoupled counting and later addition of the individual results. Perhaps it is better in this case a simplified design. Remove the Worker class and rename Master as CountInterLocked: each thread executes Compute, this method increments the shared resource (result) if the BitCoin value is above the threshold. The Index is managed with `lock` as in the previous case. When all the threads end, ComputeTimes returns result directly (you can use a read only property instead). There is no need to accumulate the per thread result.



Exercise 2. This exercise is about find and correct racing conditions, deadlocks and other nastiness in the provided source code. It is an implementation of the Sleeping Barber attributed to Edsger Dijkstra that can be found in Tanembaum OS book. Indeed, it is this implementation (actually the link is broken) translated to C#.

Consider a barber shop with just one barber. There is a barber chair where customers hair is cut and n chairs in the waiting room for the customers. If there are no customers waiting, the barber takes a pillow and sleeps in the barber chair. The path from each customer house to the barber shop takes a variable amount of time for each of them. Initially the barber is sleeping. For the sake of simplicity lets ignore the arrival order. When the first customer arrives, he will awake the sleeping barber and take the pillow away from him. Because the barber is not cutting the hair to anybody, the customer sits in the barber chair and fastens the chair belt. When the cut is finished, the barber releases the barber chair belt and then the customer gets out of the chair and leaves the barber shop. If more customers arrive while the barber is working, they sit on a free chair if any or simply wait outside.

In the provided source code managing semaphore related statements are missing. If you run the code as is, it is easy to see abnormal situations. The lines where the statements are missing are marked with empty comments. The statements can be semaphoreIdentifier.WaitOne() or semaphoreIdentifier.Release(), the first one is used in situations like “wait”. The second “take away from”, “release”, “get out of” or “leave”. Semaphore state can be interpreted as follows, zero value is like a red light, thus the thread waits; greater than zero value is green, the thread continues. Example: if the pillow semaphore value is zero, the barber thread waits.

a.- Add the statements needed in order to solve the race conditions, deadlocks and insanity. You can read the linked wikipedia article and the semaphore descriptions in the provided source code. Some of the situations usual in a barber shop (in the order as found in the source code) and the involved characters are:

- The barber grabs a pillow and waits sleeping until somebody takes his pillow away from him.
- The barber releases the chair belt when a cut is finished.
- A customer waits until there are free chairs in the waiting room.
- A customer waits until there is no other customer in the barber chair.
- A customer leaves the waiting room.
- A customer takes the pillow away from the barber.
- A customer waits while the barber chair belt is fasten.
- A customer gets out of the barber chair.
- Fibally, when the work is finished (al the customer threads are finished) the barber partner (Main) takes the barber pillow away from him.

Run your code, check if everything is fine, look for pitfalls before doing the next exercise.

b.- Identify sections in the code that need a mutex: it shouldn't happen that a customer tries to awake the barber while the previous customer hasn't left the barber shop yet (but he has get off the chair already). That is, for a customer to leave the waiting room, the previous one must leave the barber shop first or ,in other words, only one customer is allowed in the room where the barber chair is. Run the application, see if the displayed message look fine.

NOTE: Perhaps there is another resource (in the computer) that needs to be managed with a mutex too, think carefully about this, it is really cumbersome.

Exercise 3. In the provided source code you can find a very simple class with only a get/set. The set is coded in such a way that only updates the single int attribute if the new value is higher than the old one. In main, this class is used in a Parallel.For loop. Check that the results sometimes are incorrect: the final value should be the max of the int numbers in the array. Solve the issues using this approach.