MONITORS – FILE COPIER

1 OBJECTIVES

The main goals of this assignment are to learn and implement the follow concepts of multithreading:

- Monitors as synchronization mechanisms.
- Bounded buffer as shared resource for communication between multiple threads.
- Producer/Consumer and Writer/Reader patterns

2 DESCRIPTION

2.1 **File Copier**: an application that reads the contents of a file from a source into the memory, let the user manipulate the text (search and replace) and copy to a destination file. The acts of reading, modifying and writing to file must take place by threads at the same time. Although it is possible to implement the Producer/Consumer model for the solution, a Writer/Reader scenario is recommended.

This should be GUI-based and programmed in either C #or Java (or C++). The GUIs for both C# and Java will be provided.

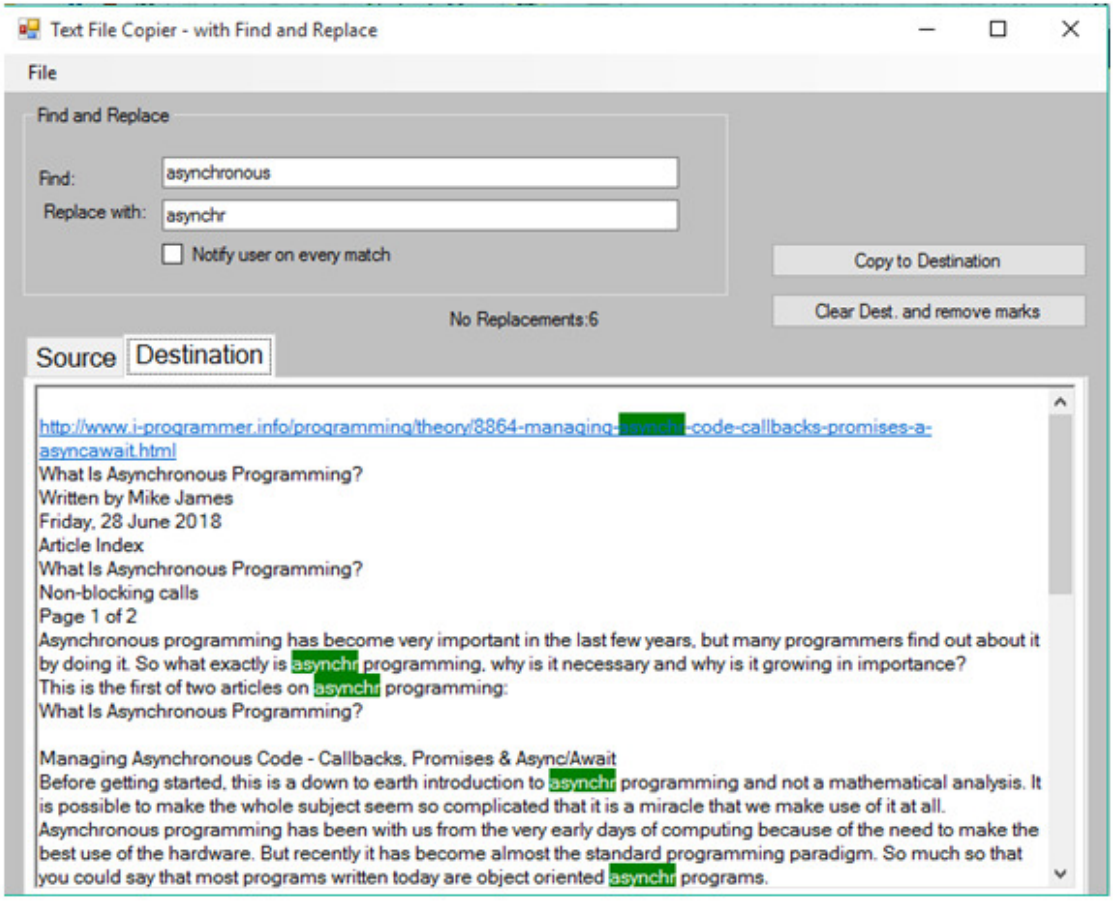
3 FILE COPIER

Via the File menu, the uses loads a text file from the file system for copying to a target destination. The contents of the file is displayed in the GUI as in the image so the user can modify the text before copying. The find and replace feature is administered by the program through two input boxes as in the suggested GUI image. The user inputs a search string and a substitute string, the program highlights all

occurrences and performs the replacement with or without the user's confirmation (see the Notify user option on the GUI). Note that highlighting and user conformation is optional. The user can of course copy the file without any modifications.

A suggested design of the GUI is presented here. In this design, two rich textboxes each as page in a **TabControl** (C#) are used, the **Source** tab and the **Destination** tab. The Source tab is intended to contain text from the original file plus all changes that the user brings before copying (optional). The **Destination** tab is to contain the modified copy of the file.

Every time the user clicks the button "**Copy to Destination**", the contents of the **Source** tab is copied to the **Destination** tab. Although the search strings are highlighted in the **Source** tab, the changes are seen only in the **Destination** tab, as illustrated in the image here.



3.1 Write an application that solves the problem as described above.

3.2 The copying can be carried out by three separate threads, one for reading, one for modifying, and one for writing. The thread synchronization can be made with the aid of monitors, allowing only one thread at a time working with the string buffer.

3.3 Use a bounded buffer and write a class for that. The size of the buffer can be sent to this class as a constructor parameter. Assuming that the bounded buffer is created with say 10 (or 15) elements, this means that the writer can write in 10 (or 15) positions before it has to wait. The number of positions in this buffer must be less than the number of strings in the input text file. Each position in the buffer has one of three status: **Empty**, **New** and **Checked**. At the start, all positions are Empty.

3.4 The writer thread can only write to a location witch is marked with the status value **Empty**. The modifier thread then checks the written string(s) for a find-and- replace operation of the given substring at a position in which the saved element (string) contains the search string, and if found, it makes the modification at that position, changing the status of this position to **Checked**.

3.5 The reader thread can only read a position having the status **Checked**, and when reading is done it resets the status to **Empty**, allowing the writer to proceed. The buffer also holds three flags (**int**) to remember the positions in the buffer,i.e. one position pointer to each operation, a write pointer (writePos), a read pointer (readPos) and one modify pointer (findPos). Create an enum **Status**, so you can then use an array of this enum to save the status of each element. All elements have the status **Empty** at start.

```
public enum Status
{
    Empty,
    Checked,
    New
}
```

3.6 A first writer puts (writes) a string in the element at position **writePos** and meantime update the status array with the value **New**. It then advances this pointer to next position using the algorithm **(pos+1) % bufferSize**. Now the modifier thread can modify this location, while the writer continues until it reaches a position that is not empty. After the modifier has done its task at the current position, it changes status to **Checked** for that position and advances its position pointer. Then the Reader is allowed to read the string and stores it in an output list, setting the status again to **Empty**, letting the writer write again.

Note: In this assignment only one writer and one reader is included but if you would like to work with a more complicated case, you can consider multiple writers and multiple readers.

A help document is available on the module.

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 Comment your code by using comments. It suggested (not required) to use xml headers on classes and methods in C#, use JavaDoc on java files.

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

4.5 Test your application carefully before submitting.

5 GRADING AND SUBMISSION

Submit your solution in Canvas and show your work to your lab-supervisor as before..

Good Luck!

Farid Naisan,
Course Responsible and Instructor