# INTRODUCTION

This example was inspired by a recent job interview question. This was the original question:

*Write a function that takes a source directory path, a search string, and a destination filename.*

*The function should then open all of the files in the given directory in parallel (using async technique of your choice).*

*Find lines that have the search text in them.*

*Extract and output all lines with that search text in a file with the given output filename.*

*At the end of execution, the function should output the number of files it processed, the number of lines it found the search text in, and the number of occurrences of that search term (don't assume once per line).*

# SUMMARY

The purpose of this application is to demonstrate Asynchronous Processing.

The application starts a task for each file in a source folder. Each task is passed a file name, a string pattern, a root path and a delay time. The task searches each line in the file looking for a pattern match. If a match occurs, the line and the number of matches in that line are added to an output object. The delay is for demonstration and is used to confirm that the tasks are actually being executed asynchronously. The delay is not necessary.

When all of the tasks complete, the results object of all of the tasks are returned as a collection.

The lines matched are written to a single output file. The total number of files processed, the total line matches count and the total pattern matches counts are written to the console.

# CONDITIONS

1. This is a simple C# console application. All of the code is included in a single file. Running the application is as simple as creating a console application in Visual Studio and replacing the Program.cs file with the example code.
2. The structure of this application is intentionally simple in order to not obfuscate the Async example.
3. There is no exception handling in this application, such as TRY/CATCH/FINALLY or USE(…){…}. This is intentional because it would obfuscate the Async example.

# APPLICATION FLOW

Main( )

Main is responsible for preparing to run MatchCountFolderAsync( ). Main( ) does these things:

* Sets the rootPath and the output file name.
* Deletes the output file name generated from a previous run.
* Creates a pattern list and delay list, which will be used to test the asynchronous behavior.
* Builds an input argument list which is passed to MatchCountFolderAsync( ).

MatchCountFolderAsync( )

The primary application flow is contained in MatchCountFolderAsync( ). MatchCountFolderAsync( ) does these things:

* Starts a Task for each file in the argument list.
* Each task calls the MatchCountFileAsync( ) method.
* When all tasks are complete, the results are output to a results collection. See MatchCountFile( ) for the specific results returned.
* FsWriteMatched( ) outputs the matched string arrays in the results collection.
* The match line count and match count are summed by iterating over the results collection.
* Console statements output the total number of files processed, the summed total match line count and the summed total match count.

MatchCountFileAsync( )

This method encapsulates the work performed by the task. This method processes a single file. The method returns this information:

* Each line that was matched in a string array.
* The total number of lines that were matched.
* The total number of matched (there will be cases where the pattern will occur more than once).

# HELPER METHODS

FetchTestFiles( )

This method creates a collection of all the files in the rootPath. The rootPath is passed as an argument.

CleanUp( )

This method deletes the output file, if it exists.

BuildArgs( )

This method builds a collection of structs that contain the following properties:

* Delay: This value is randomly generated from a delay list collection that is passed as an argument.
* Pattern: This value is randomly generated from a pattern list collection that is passed as an argument.
* FileName: This value represents the name of one file in the source folder.

# FINAL THOUGHTS

This application could be used to aggregate log file data, in multiple files, in a single output file. When Dependency Injection is added, different filters would be applied to specific file names or types.

This application makes extensive use of Tuple<>. This is convenient because it eliminates the need to declare classes just for output results. In retrospect, creating some classes may improve readability.

The application uses Console statements for demonstrating that the files are being processed asynchronously. These lines could be removed without harm. The delays are also used to demonstrate the asynchronous processing. The delays are not necessary and they could be removed without harm.

The biggest challenge in this application was returning results from the tasks. This deserves serious planning before writing an asynchronous application.

There is a potential issue, with this design, if the number of files to be processed is high. All results are aggregated into a collection after all tasks complete. This has the potential of consuming large amounts of memory. The CopyToAsync( ) and WriteAsync( ) would be likely solutions.

My goal of publishing this document and the code is two fold:

* Demonstrate my understanding of Asynchronous processing in a working application.
* Help people trying to understand Asynchronous processing with a simple working example.