Tasks and Concurrent Data Structures

Estimated time for completion: 45 minutes

Overview:

In this lab you will use the new Task API and Concurrent data structures to perform processing of a list of Dow Jones trading day statistics

Goals:

- Create tasks to perform asynchronous processing
- Use a concurrent queue and blocking collection to coordinate producer and consumer tasks
- Implement cancellation to allow the controlling application to cancel the processing

Lab Notes:

N/A

Part 1: Examine the starting project

In this part of the lab you will look at the initial project to familiarize yourself with the code that you will be driving

1. In the base directory for this lab you will find a file called **DowJones.csv**. This file contains summary information for every day of trading on the US stock market since 1928. Open this file to examine the contents. There is a header line and then each day is in csv format. This is the data that you will be processing.
2. Open the solution in the before directory
3. Examine `Main` in **program.cs**.
    a. This creates a `TradeDayProcessor` passing in the number of consumers to use, the path to the file and the test to perform on the data. The test used here is days where that the market closed more than 5% up.
    b. It starts processing by calling the processor's `Start` method.
    c. It gets the results of the processing by calling `GetMatchingCount`
    d. The time taken to perform processing and the number of trade days that matched the test are printed out.
4. Examine the `TradeDayProcessor` class in **TradeDayProcessor.cs**
    a. The constructor stored the data passed in
    b. There are stubs for all of the methods you will be implementing
    c. There are two methods for processing the file. `ParseTradeEntry` turns a file line into a `TradeDay` object. `ReadStockData` is an iterator method that iterates each

line in the file projecting a `TradeDay` object (`TradeDay` is defined in **TradeDay.cs**)

Part 2: Create the Generator Task

In the second part of the lab you will create a task that gets the data from the file and puts the data into a concurrent queue (via a blocking collection decorator) for the consumers to process

1. Open **TradeDayProcessor.cs**
2. Create a member variable called `col`. Make the member variable a `BlockingCollection<TradeDay>`. Use a field initializer to create the collection wrapping it round a `ConcurrentQueue<TradeDay>` which will provide the underlying storage.

```
BlockingCollection<TradeDay> col =
          new BlockingCollection<TradeDay>(new ConcurrentQueue<TradeDay>());
```

3. Create a member variable called `generateTask` to hold a reference to the generator task. This task will not return any result so the type of this variable is the non-generic version of `Task`. You do not need to initialize this variable for the time being.
4. We will now implement the `GenerateTradeDays` method. This is the method that will be driven by the `generateTask`.
    a. Consume the iterator returned by `ReadStockData` in a `foreach` loop and add each `TradeDay` to the blocking collection
    b. Temporarily write out the trade day to the console do you see some output when we run the project at the end of this section
    c. After the `foreach` loop call `CompleteAdding` on the blocking collection to inform any consumer that there is no more data

```
foreach (var item in ReadStockData())
{
    col.Add(item);
    Console.WriteLine(item);
}

col.CompleteAdding();
```

5. Now go to the `Start` method where we will create and start the `generateTask`
    a. Create a new `Task` wrapping it round the `GenerateTradeDays` method and assign it to the `generateTask` member variable
    b. Call `Start` on the `generateTask`
6. To see any results you will have to stop the process exiting. For the time being add a `Console.ReadLine` at the end of `Main`
7. Compile and start the project and confirm that the trade days are written to the console
8. Remove the temporary output you created in part 4b and the `Console.ReadLine` you added in step 6

Part 3: Creating the Consumer Tasks

The third part of the lab gets you to implement the consumer side of the trade day processing. You will be creating as many consumers as specific in the constructor of the TradeDayProcessor

1. Create and initialize a member variable called `consumingTasks` consumer tasks. The consumer tasks will return the number of trade days they find that match the test passed into the `TradeDayProcesor` constructor, Therefore this variable should be of type `List<Task<int>>`.
2. Locate the `ConsumeTradeDays` method. We will now implement this.
    a. Create a `foreach` loop to consume the iterator returned by the `GetConsumingEnumerator` method on the blocking collection. This method will block waiting for items to appear in the collection and will return them when appear. It is safe to call on multiple threads as the underlying concurrent queue is thread safe.
    b. Inside the loop put a `Thread.Sleep(5)` just to make the consuming work more expensive
    c. Test the current `TradeDay` using the `Predicate<TradeDay>` passed into the `TradeDayProcessor` constructor and increment the `matchingCount` local variable.
    d. Also if the `TradeDay` matches the test write it out to the console
    e. Note that the `matchingCount` is returned from this method and will be available as the `Result` of the task

```
foreach (var item in col.GetConsumingEnumerable())
{
    Thread.Sleep(5);

    if (test(item))
    {
        matchingDays++;
        Console.WriteLine(item);
    }
}
```

3. Go to the Start method. You will now create and start the consumer tasks
    a. Create a `for` loop between 0 and `numConsumers` (the number of consumers passed into the `TradeDayProcessor` constructor)
    b. Inside the loop, create a `Task<int>` wrapped round the `ConsumeTradeDays` method
    c. Add the new task into the `consumingTasks` list you created earlier
    d. Call `Start` on the new task

```
for (int i = 0; i < numConsumers; i++)
{
    var task = new Task<int>(ConsumeTradeDays);
    consumingTasks.Add(task);
    task.Start();
```

```
}
```

4. Finally you will implement the `GetMatchingCount` method. Here you will wait on the spawned tasks to complete. However, as you have learned, exceptions in tasks get thrown when you `Wait` on a task so this code will be in a `try` … `catch` block.
    a. Inside the try block create a List<Task> local variable and add the generator and consumer tasks into this list.
    b. Wait for all the tasks to complete by calling Task.WaitAll
    c. Once the consumer tasks are complete iterate across all of them summing their individual Result properties. This will produce the full total of matches to the supplied Predicate<TradeDay>
    d. Return the summed value
    e. Now you will implement the catch block
    f. Iterate through the InnerExceptions collection of the AggregateException. Remember to call Flatten on the exception in case there are nested AggregateExceptions
    g. Write out the exception message of each exception
    h. Return -1 to the caller as the processing did not complete

```
public int GetMatchingCount()
{
    try
    {
        List<Task> tasks = new List<Task>();
        tasks.Add(generateTask);
        tasks.AddRange(consumingTasks);

        Task.WaitAll(tasks.ToArray());

        int total = 0;
        foreach (var item in consumingTasks)
        {
            total += item.Result;
        }

        return total;
    }
    catch (AggregateException x)
    {
        foreach (var item in x.Flatten().InnerExceptions)
        {
            Console.WriteLine(item.Message);
        }

        return -1;
    }
}
```

5. Run and test the application.

6. Increase the number of consumers and examine how this affects performance. Increase this number in 10s. You should eventually see that after the performance of the processing starts to decrease. Parallel programming is not magic and there are many factors that affect the performance of an algorithm. The Thread.Sleep in the consuming task in this lab allows the lab to show how multiple tasks can be used in the producer/consumer pattern but does not give a realistic processing model. The session on Parallel Processing will address parallelization of algorithms.

Part 4: Implementing Cancellation

In the final part of the lab you will implement functionality to enable you to cancel the processing. The main thread will be blocked in a Console.ReadLine therefore we will need to spawn a cancellation task.

1. In `Main`, in **program.cs**, create a local variable of type `CancellationTokenSource`. This is the type at the root of the new .NET 4.0 cancellation framework.
2. Create a `Task` wrapping the `Canceller` method. This needs to be passed the `CancellationTokenSource` as a parameter.
3. `Start` this new task

```
CancellationTokenSource src = new CancellationTokenSource();
Task cancelTask = new Task(Canceller, src);

cancelTask.Start();
```

4. We will now implement the `Canceller` method
    a. Cast the passed object to a `CancellationTokenSource`
    b. Use a `Console.WriteLine`/`Console.ReadLine` pair to ask the user to press enter if they want to cancel and wait for the user to press enter
    c. If the user presses enter (`Console.ReadLine` returns) then call `Cancel` on the `CancellationTokenSource`
    d. Note that if the user does not press enter the task will still end; the main thread will terminate and the cancellation task is running on a background thread

```
private static void Canceller(object o)
{
    CancellationTokenSource src = (CancellationTokenSource)o;

    Console.WriteLine("press enter to cancel");
    Console.ReadLine();

    src.Cancel();
}
```

5. Now we need to change the TradeDayProcessor to respect the cancellation. The processing will need to be able to see a CancellationToken and so you will pass this to the Start method of the TradeDayProcessor
6. Create a CancellationToken member variable to store the passed cancellation token
7. How you need to make some changes to the Start method

a. Store the passed cancellation token in the created member variable
b. Pass the cancellation token into the constructors for the generator and consumer tasks

```
public void Start(CancellationToken token)
{
    cancellationToken = token;

    generateTask = new Task(GenerateTradeDays, cancellationToken);
    generateTask.Start();

    for (int i = 0; i < numConsumers; i++)
    {
        var task = new Task<int>(ConsumeTradeDays, cancellationToken);
        consumingTasks.Add(task);
        task.Start();
    }
}
```

8. Now we will amend the `GenerateTradeDays` method to respect cancellation
    a. In the `foreach` loop add a call to `ThrowIfCancellationRequested` on the `cancellationToken` each time through the loop to enable the loop to break out if requested to cancel

```
foreach (var item in ReadStockData())
{
    cancellationToken.ThrowIfCancellationRequested();
    col.Add(item);
    Console.WriteLine(item);
}
```

9. Now you need to change `ConsumeTradeDays` to also respect cancellation. This must be done in two places
    a. Pass the `cancellationToken` into the `GetConsumingEnumerator` method. This allows the method to throw an exception if cancellation is requested while it is blocking
    b. Inside the processing loop call `ThrowIfCancellationRequested` on the `cancellationToken` to allow the loop to break out if cancellation is requested

```
foreach (var item in col.GetConsumingEnumerable(cancellationToken))
{
    Thread.Sleep(5);

    cancellationToken.ThrowIfCancellationRequested();
    if (test(item))
    {
        matchingDays++;
        Console.WriteLine(item);
    }
}
```

10. Finally, the call to Start in Main no longer matches the amended signature. Change the call to pass the Token from the CancellationTokenSource
11. Now compile and test your code and confirm you can cancel the processing