Lab 03 - Parallel Practices

Objectives

This lab adds parallel methods to a web application. This will look at different methods of adding parallelism as well as concurrency issues to watch out for (and how to fix them). In particular, we will see the importance of using collections that are thread-safe.

Application Overview

The "Starter" folder contains the code files for this lab.

Visual Studio 2022: Open the "UnderstandingAsync.sln" solution.
Visual Studio Code: Open the "Starter" folder in VS Code.

Note: The lab also contains a "Completed" folder with the finished solution. If you get stuck along the way or have issues with debugging, take a look at the code in the "Completed" folder for guidance.

This solution contains a number of projects that mirror the demo code for the workshop. This lab concentrates on the "Parallel.UI.Web" project (with some bonus challenges in the "Parallel.UI.Desktop" and "Parallel.Basic" projects).

The "Parallel.UI.Web" project contains a web application that makes multiple service calls.

The "People.Service" project contains the service called by the other "Parallel.UI" projects.

The other projects (including "TaskAwait.Library" and "TaskAwait.Shared") contain supporting classes and methods.

Running the Application

If you are using Visual Studio 2022, start by re-building the solution (from the "Build" menu, choose "Rebuild Solution"). If you are using Visual Studio Code, you do not need to worry about this step; this is mainly to get the Visual Studio environment in sync.

1. Start the service.

For this lab, the service must be running. You can start the service from the command line and leave it running the entire time.

To start the service, open the "People.Service" folder in your favorite command-line tool (cmd, Terminal, PowerShell, etc.). One way to do this in Windows 11 is to open the "People.Service" folder in File Explorer, and then right-click a blank space in the folder. There should be an option such as "Open in Terminal" or "Open in PowerShell".

Alternately, you can open PowerShell or Terminal (if installed) and then navigate to the service folder using the "cd" (change directory) command.

```
PS C:..\Lab05\Starter\People.Service>
```

Type "dotnet run" to start the service.

```
PS C:..\Lab05\Starter\People.Service> dotnet run
Building...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:9874
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:..\Lab05\Starter\People.Service\
```

You can check the service is running by opening the following in your browser: http://localhost:9874/people.

The specific endpoint used by these samples uses an "ID" parameter to get an individual record. For example, http://localhost:9874/people/3 returns the "Turanga Leela" record.

> *Note: The service has a 1 second delay before returning a record. This is to simulate latency and to make it easier to see the parallel code working.*

2. Run the web application.

In Visual Studio 2022: Set the "Parallel.UI.Web" application to the startup project. Press "F5" or use the "Debug" menu to start debugging.
In Visual Studio Code: Set the "Parallel.UI.Web" application to debug in a browser. If you have not done this before, then I would recommend starting the web application from the command line (using the same method as above to start the service).
Command Line: Similar to starting the service, open your command line tool to the "Parallel.UI.Web" folder and type "dotnet run" to start the application. You will see "Now listening on: http://localhost:5000".

Open a browser to http://localhost:5000, and you will see options to "Get People" using various methods.

Click "Get People using Await (not parallel)" to make sure everything is working.

> *Note: Since this code is not running in parallel, it will take 10 seconds for the results to show.**

Lab Goals

Two of the methods need to be implemented:

- Get People using Task (parallel)
- Get People using ForEachAsync (parallel)

These will use Tasks with Continuations, and a Parallel.ForEachAsync loop, respectively.

Current Classes

The changes will be made in the "PeopleController" class of the web application. There are already actions stubbed out for the methods to be implemented.

Parallel.UI.Web/Controllers/PeopleController.cs:

```
// Task w/ Continuation (runs parallel)
public async Task<IActionResult> WithTask()
{
    await Task.Delay(1);
    return Content("This has not been implemented");
}

// Parallel.ForEachAsync (runs parallel)
public async Task<IActionResult> WithForEach()
{
    await Task.Delay(1);
    return Content("This has not been implemented");
}
```

In addition, the non-parallel "using Await" method is available for reference:

```
// Await (runs sequentially)
public async Task<IActionResult> WithAwait()
{
    ViewData["Title"] = "Using async/await (not parallel)";
    ViewData["RequestStart"] = DateTime.Now;
    try
    {
        List<int> ids = await reader.GetIdsAsync();
        List<Person> people = new();

        foreach (int id in ids)
        {
            var person = await reader.GetPersonAsync(id);
            people.Add(person);
        }

        return View("Index", people);
    }
    catch (Exception ex)
    {
        List<Exception> errors = new() { ex };
        return View("Error", errors);
    }
    finally
    {
        ViewData["RequestEnd"] = DateTime.Now;
    }
}
```

Note: A walkthrough of this method is included in the Step-by-Step instructions below.

Hints

- The "ViewData" and error handling blocks can be re-used from the "WithAwait" method.
- The "foreach" loop of the "WithAwait" method can be re-used for the task/continuation code (with changes to the loop body).
- The body of the "foreach" loop of the "WithAwait" method can be used for the Parallel.ForEachAsync code.
- Consider using concurrent collections in the code.
- Consider changing the parallel options of the Parallel.ForEachAsync code.

BONUS CHALLENGE: Additional Concurrent Items

- In the "Parallel.UI.Desktop" project, fix the issue with the "Parallel.ForEachAsync" method. A hint is included in the comments above the method.
- In the "Parallel.Basic" project, remove the need for the "lock" statements by making the "DisplayPerson" method thread-friendly.

If you want more assistance, step-by-step instructions are included below. Otherwise, if you'd like to challenge yourself, **STOP READING NOW**

Review of the "WithAwait" Method

Start by reviewing the "WithAwait" method. Here is a walkthrough:

```
    // Await (runs sequentially)
    public async Task<IActionResult> WithAwait()
    {
        ViewData["Title"] = "Using async/await (not parallel)";
        ViewData["RequestStart"] = DateTime.Now;
        try
        {
            // bulk of code not shown
            return View("Index", people);
        }
        catch (Exception ex)
        {
            List<Exception> errors = new() { ex };
            return View("Error", errors);
        }
        finally
        {
            ViewData["RequestEnd"] = DateTime.Now;
        }
    }
```

1. The "ViewData" elements are used for the UI. "Title" shows up at the top of the page, and the "RequestStart" and "RequestEnd" times show up at the bottom. The times give us an idea of how long it took to generate the page.

2. The "try" block show the success path.

```
try
{
    // bulk of code not shown
    return View("Index", people);
}
```

This will return the "Index" view and use the "people" collection for the data. (We will look at the details of this part in just a bit.)

3. The "catch" block shows the error path.

```
catch (Exception ex)
{
    List<Exception> errors = new() { ex };
    return View("Error", errors);
}
```

The "Error" view is set up to take a collection of exceptions for its data. This catch block puts the exception into a new list and then returns the "Error" view. (The "Error" view is set up to show multiple exceptions, so it takes a list of exceptions as data.)

4. The "finally" block makes sure that the end time is recorded.

```
finally
{
    ViewData["RequestEnd"] = DateTime.Now;
}
```

5. Here is the body of the "try" block:

```
try
{
    List<int> ids = await reader.GetIdsAsync();
    List<Person> people = new();

    foreach (int id in ids)
    {
        var person = await reader.GetPersonAsync(id);
        people.Add(person);
    }
```

```
            return View("Index", people);
        }
```

- The call to "GetIdsAsync" gets a list of IDs that we can loop through.
- The "people" list is used to hold the data that is passed eventually passed to the "Index" view.
- The "foreach" loop goes through each of the IDs.
- Inside the loop, "GetPersonAsync" gets an single Person record based on the ID. Since this code is awaited, it will take at least 1 second for each iteration of the loop.
- After the "Person" record is returned, it is added to the "people" list.
- Finally, we return the "Index" view and pass the complete "people" list as a data parameter.

Parts of this method can be used to fill in the tasks/continuations method and the Parallel.ForEachAsync method.

Parallel using Tasks and Continuations: Step-By-Step

1. Remove the place-holder code in the "WithTask" method.

```
    public async Task<IActionResult> WithTask()
    {

    }
```

2. Copy the contents of the "WithAwait" method into the body of the "WithTask" method. Change the 'ViewData["Title"]' value on the first line to "Using Task (parallel)".

```
    public async Task<IActionResult> WithTask()
    {
        ViewData["Title"] = "Using Task (parallel)";
        ViewData["RequestStart"] = DateTime.Now;
        try
        {
            List<int> ids = await reader.GetIdsAsync();
            List<Person> people = new();

            foreach (int id in ids)
            {
                var person = await reader.GetPersonAsync(id);
                people.Add(person);
            }

            return View("Index", people);
        }
```

```
        catch (Exception ex)
        {
            List<Exception> errors = new() { ex };
            return View("Error", errors);
        }
        finally
        {
            ViewData["RequestEnd"] = DateTime.Now;
        }
    }
```

This lets us use the "ViewData" elements as well as the "try/catch/finally" block that we need.

> If we run the application at this point, the "Get People using Task" link will work the same as the "Get People using Await" link. It will take at least 9 seconds to complete.

The next steps are to modify the contents of the "try" block.

3. Remove the "await" keyword from the call to "GetPersonAsync" method.

```
        try
        {
            List<int> ids = await reader.GetIdsAsync();
            List<Person> people = new();

            foreach (int id in ids)
            {
                var person = reader.GetPersonAsync(id);
                people.Add(person);
            }

            return View("Index", people);
        }
```

At this point, the code will not compile. If you hover the cursor over the "var" on the "person" variable, you will see that this is now a "Task<Person>" rather than a "Person".

4. Change "var person" to be a "Task<Person> personTask".

```
    foreach (int id in ids)
    {
        Task<Person> personTask = reader.GetPersonAsync(id);
        people.Add(person);
    }
```

This gives our variable a better name and shows the type explicitly.

5. Add a continuation to the "personTask".

```
    foreach (int id in ids)
    {
        Task<Person> personTask = reader.GetPersonAsync(id);
        personTask.ContinueWith(task =>
        {

        });
        people.Add(person);
    }
```

6. Move the step that adds to the "people" list inside the body of the continuation.

```
    foreach (int id in ids)
    {
        Task<Person> personTask = reader.GetPersonAsync(id);
        personTask.ContinueWith(task =>
        {
            people.Add(person);
        });
    }
```

7. Get the "person" value from the "Result" property of the task.

```
    foreach (int id in ids)
    {
        Task<Person> personTask = reader.GetPersonAsync(id);
        personTask.ContinueWith(task =>
        {
            var person = task.Result;
            people.Add(person);
        });
    }
```

The code will build successfully at this point. But if we run the application and click "Get People using Task", there is no data. We just get the title and the timings.

This is because we do not wait for the tasks to complete before returning the view.

8. Outside of the "foreach" loop, create a new list to hold the continuation tasks called "taskList".

```
    try
    {
        List<int> ids = await reader.GetIdsAsync();
        List<Person> people = new();
        List<Task> taskList = new();
```

```
        foreach (int id in ids)
        {
            Task<Person> personTask = reader.GetPersonAsync(id);
            personTask.ContinueWith(task =>
            {
                var person = task.Result;
                people.Add(person);
            });
        }

        return View("Index", people);
    }
```

9. Capture the task that is returned from the "ContinueWith" method and add it to the "taskList".

```
    try
    {
        List<int> ids = await reader.GetIdsAsync();
        List<Person> people = new();
        List<Task> taskList = new();

        foreach (int id in ids)
        {
            Task<Person> personTask = reader.GetPersonAsync(id);
            Task continuation = personTask.ContinueWith(task =>
            {
                var person = task.Result;
                people.Add(person);
            });
            taskList.Add(continuation);
        }

        return View("Index", people);
    }
```

10. Use "Task.WhenAll" to wait for all of the continuations to complete before returning the view.

```
    try
    {
        List<int> ids = await reader.GetIdsAsync();
        List<Person> people = new();
        List<Task> taskList = new();

        foreach (int id in ids)
        {
            Task<Person> personTask = reader.GetPersonAsync(id);
            Task continuation = personTask.ContinueWith(task =>
            {
                var person = task.Result;
```

```
                people.Add(person);
            });
            taskList.Add(continuation);
        }

        await Task.WhenAll(taskList);
        return View("Index", people);
    }
```

Now when we run the application and click "Get People with Task", we get data back. Notice, that it now takes 1 second instead of 9 seconds since we are getting the data in parallel.

BUT THERE'S A PROBLEM

Try refreshing the screen several times. There should be a total of 9 records returned, but sometimes you may get fewer than 9 records. What's happening here?

> We are running into a concurrency problem. The List<T> type is not thread-safe. This means that if we try to add things to a list from multiple threads, some of the "Add" calls may fail. In our case, this results in lost data.

To fix this, we can use one of the concurrent collections included in the .NET libraries. The "BlockingCollection<T>" class is closest to "List<T>" and will work for our application.

11. Change the "List<Person>" type for "people" to "BlockingCollection<Person>".

```
    List<int> ids = await reader.GetIdsAsync();
    BlockingCollection<Person> people = new();
    List<Task> taskList = new();
```

Now when we run the application, we will consistently get all 9 records back, even with multiple refreshes.

Here is the final method:

```
    public async Task<IActionResult> WithTask()
    {
        ViewData["Title"] = "Using Task (parallel)";
        ViewData["RequestStart"] = DateTime.Now;
        try
        {
            List<int> ids = await reader.GetIdsAsync();
            BlockingCollection<Person> people = new();
            List<Task> taskList = new();

            foreach (int id in ids)
            {
                Task<Person> personTask = reader.GetPersonAsync(id);
                Task continuation = personTask.ContinueWith(task =>
                {
                    var person = task.Result;
```

```
                    people.Add(person);
                });
                taskList.Add(continuation);
            }

            await Task.WhenAll(taskList);
            return View("Index", people);
        }
        catch (Exception ex)
        {
            List<Exception> errors = new() { ex };
            return View("Error", errors);
        }
        finally
        {
            ViewData["RequestEnd"] = DateTime.Now;
        }
    }
```

A few notes about where we need to use a concurrent collection:

- For the list of IDs, we do **not** need a concurrent collection. This is because we add to the list all at the same time, and we read from the list sequentially (in the "foreach" loop).
- For the list of people, we **do** need a concurrent collection since we add to the list from multiple threads (i.e., the simultaneously running continuations).
- For the list of tasks, we do **not** need a concurrent collection since we add to the list sequentially (in the "foreach" loop).
- For the error list, we do **not** need a concurrent collection since this is not part of the parallel process.

This completes the code that uses tasks and continuations.

Parallel using Parallel.ForEachAsync: Step-By-Step

1. Remove the place-holder code in the "WithForEach" method.

```
    public async Task<IActionResult> WithForEach()
    {

    }
```

2. Copy the contents of the "WithAwait" method into the body of the "WithForEach" method. Change the 'ViewData["Title"]' value on the first line to "Using ForEachAsync (parallel)".

```
    public async Task<IActionResult> WithForEach()
    {
        ViewData["Title"] = "Using ForEachAsync (parallel)";
```

```
        ViewData["RequestStart"] = DateTime.Now;
        try
        {
            List<int> ids = await reader.GetIdsAsync();
            List<Person> people = new();

            foreach (int id in ids)
            {
                var person = await reader.GetPersonAsync(id);
                people.Add(person);
            }

            return View("Index", people);
        }
        catch (Exception ex)
        {
            List<Exception> errors = new() { ex };
            return View("Error", errors);
        }
        finally
        {
            ViewData["RequestEnd"] = DateTime.Now;
        }
    }
```

This lets us use the "ViewData" elements as well as the "try/catch/finally" block that we need.

Rather than changing the contents of the "foreach" loop, we will keep the contents and replace it with a "ForEachAsync" loop.

3. Remove the "foreach" loop and add a "ForEachAsync" method call.

```
    try
    {
        List<int> ids = await reader.GetIdsAsync();
        List<Person> people = new();

        await Parallel.ForEachAsync(
            ids,
            async (id, token) =>
            {
                var person = await reader.GetPersonAsync(id);
                people.Add(person);
            });

        return View("Index", people);
    }
```

A few notes:

- The call to "Parallel.ForEachAsync" is awaited. This means that the view will not be returned until after all of the loop iterations are complete.
- The first parameter of "ForEachAsync" is the collection to loop through. In this case, it is the list of IDs.
- The second parameter is a delegate (a lambda expression in this case) that will run for each iteration of the loop.
- The lambda expression is marked as "async" because we use "await" in the body of the expression.
- The first parameter of the lambda expression ("id") is the current element. This is similar to a "foreach" item that we can use in the lambda expression body.
- The second parameter of the lambda expression ("token") represents an optional cancellation token.

4. Since we do not use the cancellation token, we can replace "token" with a discard.

```
    await Parallel.ForEachAsync(
        ids,
        async (id, _) =>
        {
            var person = await reader.GetPersonAsync(id);
            people.Add(person);
        });
```

Run the application and click "Get People with ForEachAsync". We get data, but not all of the data. This is because we are still using "List" instead of "BlockingCollection" for the "people" collection.

5. Change the "people" variable to a "BlockingCollection".

```
    try
    {
        List<int> ids = await reader.GetIdsAsync();
        BlockingCollection<Person> people = new();

        await Parallel.ForEachAsync(
            ids,
            async (id, _) =>
            {
                var person = await reader.GetPersonAsync(id);
                people.Add(person);
            });

        return View("Index", people);
    }
```

> If we run the application at this point, the "Get People using ForEach" link returns all 9 records. But it takes more than 1 second to complete. How long it takes will depend on the number of cores that you computer has.

By default "ForEachAsync" will use a number of cores on your computer to determine how many things to run at the same time. In my case, I have 8 cores so it will run 8 items at a time. Since there are 9 items in the list, it will run that last item after at least one of the items finishes.

We can use a "ParallelOptions" object to specify our own settings.

6. Add a "ParallelOptions" parameter to "ForEachAsyc" to set the "MaxDegreesOfParallelism" to 3.

```
await Parallel.ForEachAsync(
    ids,
    new ParallelOptions() { MaxDegreeOfParallelism = 3 },
    async (id, _) =>
    {
        var person = await reader.GetPersonAsync(id);
        people.Add(person);
    });
```

If we run the application now (and refresh the page several times), we will see that it takes about 3 seconds to complete. This is because the loop is running 3 things at a time.

7. Change the "MaxDegreesOfParallelism" to 10.

```
await Parallel.ForEachAsync(
    ids,
    new ParallelOptions() { MaxDegreeOfParallelism = 10 },
    async (id, _) =>
    {
        var person = await reader.GetPersonAsync(id);
        people.Add(person);
    });
```

Now it only takes 1 second to complete.

> **Note on I/O-bound Operations**: This application has an "I/O-bound" operation that we run in parallel. An I/O-bound operation is one that waits for an input/output operation such as accessing a file, getting data from a database, or calling a service. Because of this, the number of concurrent operations is not dependent on the number of cores on our computer. However, in the real world, we want to be careful about the number of concurrent web calls that we run at the same time. Our example here is a bit contrived so that we can try out these different things.

> **Note on CPU-bound Operations**: The digit recognition application that is shown as part of the demo code is a "CPU-bound" operation. In that case, the operations that take time to complete are using CPU resource (a lot of simple math in our case). Because of this, we do not want the MaxDegreesOfParallelism to be greater than the number of CPU cores. The operations will still complete, but there will be a lot of switching back and forth between operations that will make performance worse. For my CPU-bound operations, I generally set the value to 1 or 2 less than the number of cores. This gives the computer a bit of space for other housekeeping operations.

This completes the ForEachAsync code that gets our data in parallel.

BONUS CHALLENGE - Fixing UI Issues: Step-By-Step

As a bonus, we will look at an issue that we need to be concerned with if we are building desktop and some types of mobile applications -- getting back to the UI thread. For this we will use the "Parallel.UI.Desktop" project.

Note: This example will only run on Windows. If you are completing this lab using macOS or Linux, you can skip to the next challenge.

1. Make sure the People.Service project is running.

Just like the web application, the desktop application gets its data from the service. Follow the instructions at the top of the lab if the service is not already running.

2. Run the desktop application.

In Visual Studio 2022: Set the "Parallel.UI.Desktop" application to the startup project. Press "F5" or use the "Debug" menu to start debugging.
In Visual Studio Code: Set the "Parallel.UI.Desktop" application to debug. If you have not done this before, then I would recommend starting the application from the command line.
Command Line: Similar to starting the service, open your command line tool to the "Parallel.UI.Desktop" folder and type "dotnet run" to start the application.

All 3 of these options will open a desktop application with multiple buttons on the left side.

3. Click through the various buttons.

- "Fetch w/ await (Non-Parallel)" takes 9 seconds to complete. You will see the items show up one at a time in the list box.
- "Fetch w/ Task (Parallel)" takes 1 second to complete. All items show at the same time.
- "Fetch w/ Channel (Parallel)" takes 1 second to complete. All items show at the same time.
- "Fetch w/ ForEachAsync (Parallel)" throws an invalid operation exception. (This is what we are here to fix).

4. Open the "MainWindow.xaml.cs" file in the "Parallel.UI.Desktop" project.

5. Locate the "FetchWithForEachAsyncButton_Click" method. Here is the "try" block from that code.

```
    try
    {
        var ids = await reader.GetIdsAsync();

        await Parallel.ForEachAsync(
            ids,
            tokenSource.Token,
            async (id, token) =>
            {
```

```
                var person = await reader.GetPersonAsync(id, token);
                PersonListBox.Items.Add(person);
            });
    }
```

Note that this code is very similar to the code in the web application that we saw earlier. The primary difference is that it adds the resulting "person" to a list box in the UI rather than returning a view.

The problem is that we can only access UI elements (like the list box) when we are on the UI thread. The "ForEachAsync" uses multiple threads to run the body of the lambda expression. None of these threads is the UI thread, so we get a runtime error.

Using a Dispatcher

One very common way of getting back to a UI thread is by using a dispatcher to marshall back to the UI thread. (That's a lot of jargon.) In this case, the desktop application has a built-in dispatcher that knows how to get back to the UI thread when we need it.

We can pass the dispatcher a delegate (usually with a lambda expression), and that delegate will run on the UI thread.

6. Use "Dispatcher.Invoke" to get back to the UI thread.

```
    await Parallel.ForEachAsync(
        ids,
        tokenSource.Token,
        async (id, token) =>
        {
            var person = await reader.GetPersonAsync(id, token);
            Dispatcher.Invoke(() => PersonListBox.Items.Add(person));
        });
```

We have wrapped the place where we use the list box in a delegate (a lambda expression).

```
() => PersonListBox.Items.Add(person)
```

The empty parentheses tell us that this lambda expression does not take any parameters.

If we pass this delegate to "Dispatcher.Invoke", it will run on the UI thread.

```
    Dispatcher.Invoke(() => PersonListBox.Items.Add(person))
```

7. Re-run the application.

Now when we click the "Fetch w/ ForEachAsync" button, the data comes back with no errors.

Depending on how many cores you have on your computer, you may see the data come back in chunks. This is the same behavior that we saw with the web application and the MaxDegreesOfParallelism.

We can fix this by adding ParallelOptions as a parameter.

8. Add "ParallelOptions" to the "ForEachAsync" call.

```
    await Parallel.ForEachAsync(
        ids,
        new ParallelOptions
        {
            CancellationToken = tokenSource.Token,
            MaxDegreeOfParallelism = 10
        },
        async (id, token) =>
        {
            var person = await reader.GetPersonAsync(id, token);
            Dispatcher.Invoke(() => PersonListBox.Items.Add(person));
        });
```

The "ParallelOptions" looks a little different this time. That is because there was also a cancellation token parameter in place. The "ParallelOptions" has a cancellation token property, so we can include both the "CancellationToken" and "MaxDegreesOfParallelism" as part of the options.

9. Build and re-run the application.

Now all of the data comes back as a single "chunk".

> Note: Marshalling back to the UI thread like this can impact performance since there is some thread-shuffling happening. Because of this, I prefer to avoid using Dispatcher.Invoke where I can (although sometimes it is the easiest approach).

As other options, using task with a continuation allows us to specify where the continuation is run using the TaskScheduler. There is still thread-shuffling happening, but it is at a different level. The option that uses channels has the least amount of thread-shuffling. In that case, the producer (getting the data) uses various threads, but then the consumer (using the data) only uses the main (UI) thread.

As usual, the best approach for a particular situation will vary.

BONUS CHALLENGE - Thread-Safe DisplayPerson Method: Step-By-Step

As another bonus, we will look at an issue that we need to be concerned with if we are building concurrent applications -- in this case, building thread-safe methods. For this we will use the "Parallel.Basic" project.

1. Make sure the People.Service project is running.

Just like the web application, this console application gets its data from the service. Follow the instructions at the top of the lab if the service is not already running.

2. Run the console application.

In Visual Studio 2022: Set the "Parallel.Basic" application to the startup project. Press "F5" or use the "Debug" menu to start debugging.
In Visual Studio Code: Set the "Parallel.Basic" application to debug. If you have not done this before, then I would recommend starting the application from the command line.
Command Line: Similar to starting the service, open your command line tool to the "Parallel.Basic" folder and type "dotnet run" to start the application.

3. Open the "Program.cs" file.

4. In the "Main" method, make sure that "Option 2" is uncommented (and the other options are commented out).

```
        // Option 1 = Run Sequentially
        //await RunSequentially(ids);

        // Option 2 = Task w/ Continuation
        await RunWithContinuation(ids);

        // Option 3 = Channels
        //await RunWithChannel(ids);

        // Option 4 = ForEachAsync
        //await RunWithForEachAsync(ids);
```

5. Run the application. The output should be similar to the following:

```
[1,2,3,4,5,6,7,8,9]
--------------
1: John Koenig
Friday, October 17, 1975
Rating: ******
--------------
2: Dylan Hunt
Monday, October 2, 2000
Rating: ********
--------------
3: Turanga Leela
Sunday, March 28, 1999
Rating: ********
--------------
5: Dave Lister
Monday, February 15, 1988
Rating: *********
--------------
6: Laura Roslin
Monday, December 8, 2003
Rating: ******
--------------
4: John Crichton
Friday, March 19, 1999
```

```
Rating: *******
--------------
7: John Sheridan
Wednesday, January 26, 1994
Rating: ******
--------------
8: Dante Montana
Wednesday, November 1, 2000
Rating: *****
--------------
9: Isaac Gampu
Saturday, September 10, 1977
Rating: ****

Total time: 00:00:01.5131592
```

6. The "RunWithContinuation" method currently has a "lock" statement.

```
    static async Task RunWithContinuation(List<int> ids)
    {
        List<Task> allContinuations = new();

        foreach (var id in ids)
        {
            Task<Person> personTask = reader.GetPersonAsync(id);
            Task continuation = personTask.ContinueWith(task =>
            {
                var person = task.Result;
                lock (ids)
                {
                    DisplayPerson(person);
                }
            });
            allContinuations.Add(continuation);
        }

        await Task.WhenAll(allContinuations);
    }
```

The "lock" around "DisplayPerson" makes sure that the method cannot be called more than once at the same time (for example, multiple threads calling "DisplayPerson" at the same time).

7. Remove the "lock" block from around "DisplayPerson".

```
    static async Task RunWithContinuation(List<int> ids)
    {
        List<Task> allContinuations = new();

        foreach (var id in ids)
```

```
        {
            Task<Person> personTask = reader.GetPersonAsync(id);
            Task continuation = personTask.ContinueWith(task =>
            {
                var person = task.Result;
                DisplayPerson(person);
            });
            allContinuations.Add(continuation);
        }

        await Task.WhenAll(allContinuations);
    }
```

8. Re-run the application. The output will be similar to the following:

```
[1,2,3,4,5,6,7,8,9]
--------------
--------------
--------------
2: Dylan Hunt
1: John Koenig
4: John Crichton
Friday, October 17, 1975
Monday, October 2, 2000
Rating: ******
Friday, March 19, 1999
Rating: *******
Rating: ********
--------------
6: Laura Roslin
Monday, December 8, 2003
Rating: ******
--------------
--------------
5: Dave Lister
Monday, February 15, 1988
Rating: *********
3: Turanga Leela
Sunday, March 28, 1999
Rating: ********
--------------
9: Isaac Gampu
Saturday, September 10, 1977
Rating: ****
--------------
8: Dante Montana
--------------
7: John Sheridan
Wednesday, November 1, 2000
Rating: *****
Wednesday, January 26, 1994
Rating: ******
```

```
Total time: 00:00:01.5105675
```

Notice that the output is mixed together. This is because of the way that the "DisplayPerson" method works.

Here is the "DisplayPerson" method:

```
    static void DisplayPerson(Person person)
    {
        Console.WriteLine("--------------");
        Console.WriteLine($"{person.Id}: {person}");
        Console.WriteLine($"{person.StartDate:D}");
        Console.WriteLine($"Rating: {new string('*', person.Rating)}");
    }
```

> The issue is that this method has a series of 4 "Console.WriteLine" calls. If this method is called multiple times at the same time, then these calls get interleaved.

One way to solve the problem is to add a "lock" around the calls so that this method can only be called one call at a time.

Another solution is to make the "DisplayPerson" call thread-safe. One way of doing this is to compose a single string and only make one "Console.WriteLine" call.

9. Create a single string with all 4 lines by using a StringBuilder.

```
    static void DisplayPerson(Person person)
    {
        StringBuilder builder = new();
        builder.AppendLine("--------------");
        builder.AppendLine($"{person.Id}: {person}");
        builder.AppendLine($"{person.StartDate:D}");
        builder.Append($"Rating: {new string('*', person.Rating)}");

        Console.WriteLine(builder.ToString());
    }
```

This appends multiple lines of text together into a StringBuilder. Then a single call to WriteLine outputs the final string.

10. Re-build and re-run the application. The output should be similar to the following:

```
[1,2,3,4,5,6,7,8,9]
--------------
4: John Crichton
Friday, March 19, 1999
Rating: *******
```

```
--------------
6: Laura Roslin
Monday, December 8, 2003
Rating: ******
--------------
7: John Sheridan
Wednesday, January 26, 1994
Rating: ******
--------------
8: Dante Montana
Wednesday, November 1, 2000
Rating: *****
--------------
3: Turanga Leela
Sunday, March 28, 1999
Rating: ********
--------------
1: John Koenig
Friday, October 17, 1975
Rating: ******
--------------
5: Dave Lister
Monday, February 15, 1988
Rating: *********
--------------
2: Dylan Hunt
Monday, October 2, 2000
Rating: ********
--------------
9: Isaac Gampu
Saturday, September 10, 1977
Rating: ****

Total time: 00:00:01.4823358
```

Since there is only a single "WriteLine" call, the "DisplayPerson" method can safely be called multiple times and still have the text come out as expected.

11. As a last step, remove the "lock" statement from the "RunWithForEachAsync" method. This method will also run correctly without needing a lock.

```
    static async Task RunWithForEachAsync(List<int> ids)
    {
        await Parallel.ForEachAsync(
            ids,
            new ParallelOptions { MaxDegreeOfParallelism = 10 },
            async (id, token) =>
            {
                var person = await reader.GetPersonAsync(id);
                DisplayPerson(person);
```

```
            });
    }
```

> When given a choice between using a thread-safe method and using a lock, I opt for the thread-safe method. By its nature, "lock" has a certain amount of overhead associated with it. By making blocking calls, it impacts the performance of the application.

If we can remove the "lock", we can also remove that performance impact. For small applications like this one, there is not much impact; however, for applications that use locks for thousands of calls, the impact can be noticeable.

This completes the look at one way to add thread safety to methods.

End of Lab 03 - Parallel Practices
