|  |
| --- |
|  |
| TPL |
| Fun with Task<T> |
|  |
|  |
|  |

|  |
| --- |
|  |

The TPL

# Overview

In this lab you’ll work with a slow PayrollService processor that simulates heavy algorithmic work. Once the lab is complete you should understand how to:

* Use Parallel
* Use Task<T>

# Part I – Benchmark and Parallelize An Algorithm

1. **Open** the **Workplace** solution in the before folder of this lab.
2. **Insepect** EmployeeList.cs and PayrollServices.cs.

The goal of part I will be to execute GetPayrollDeduction for each employee in the employee list. The GetPayrollDeduction method simulates slow work using Thread.Sleep.

1. Open **Program.cs**.
2. In the **Main** method, create a Stopwatch (from System.Diagnostics) and prepare a benchmark measurement by starting the stopwatch and writing out the elapsed time (in seconds) at the end of the method.

static void Main()

{

**var watch = Stopwatch.StartNew();**

**Console.WriteLine("Total time: {0}s", watch.Elapsed.TotalSeconds);**

}

1. Create a new **method** on Program named **ProcessEmployees.** **Invoke** the method in Main after starting the stopwatch.

static void Main()

{

var watch = Stopwatch.StartNew();

**ProcessEmployees();**

Console.WriteLine("Total time: {0}s", watch.Elapsed.TotalSeconds);

}

**private static void ProcessEmployees()**

**{**

**}**

1. In ProcessEmployees, create an instance of **EmployeeList** and save it to a local variable named employees.
2. In the ProcessEmployees method, also create an instance of PayrollServices and use the instance inside a **for** **loop** to process each employee with **GetPayrollDeduction**. Write diagnostic info to the console **before** calling the service method and **after** calling the method. The before information should include the employee ID. The after information should display the elapsed time required to process a single employee (which is a value conveniently returned by the GetPayrollDeduction method), and the employee ID.

private static void ProcessEmployees(EmployeeList employees)

{

var employees = new EmployeeList();

var service = new PayrollServices();

for (int i = 0; i < employees.Count; i++)

{

Console.WriteLine("Starting employee id {0}", employees[i].EmployeeID);

var span = service.GetPayrollDeduction(employees[i]);

Console.WriteLine("Completed process for employee id {0} took {1}s",

employees[i].EmployeeID, span);

}

}

1. **Run** the application and record the benchmark time (you might start the program from the command line and let it run while you read ahead, it might take 40 seconds).
2. Replace the for loop in ProcessEmployees with **Parallel.For**.

private static void ProcessEmployees(EmployeeList employees)

{

var employees = new EmployeeList();

var service = new PayrollServices();

**Parallel.For(0, employees.Count, i =>**

**{**

...

**});**

}

1. **Run** the application and compare the benchmark.

How did the results compare? Is the reduction in time what you expected?

1. There is also a Parallel.ForEach method available. Rewrite ProcessEmployee to use **Parallel.ForEach**.

private static void ProcessEmployees(EmployeeList employees)

{

var employees = new EmployeeList();

var service = new PayrollServices();

**Parallel.ForEach(employees, e =>**

**{**

**Console.WriteLine("Starting employee id {0}", e.EmployeeID);**

**var span = service.GetPayrollDeduction(e);**

**Console.WriteLine("Completed process for employee id {0} took {1}s",**

**e.EmployeeID, span);**

**});**

}

# Part II – Using Invoke

The TPL also include an Invoke method you can use to spin off the execution of code in a lambda expression on another thread.

1. In the Main method, replace the call to ProcessEmployees with a call to a new method - **WalkTree**.

static void Main()

{

var watch = Stopwatch.StartNew();

**WalkTree();**

Console.WriteLine("Total time: {0}s", watch.Elapsed.TotalSeconds);

}

**private static void WalkTree()**

**{**

**}**

1. In the WalTree method, create an instance of **EmployeeHierarchy** and pass it to an overloaded version of WalkTree that you’ll create. This overload of WalkTree takes a parameter of type **Tree<Employee>**.

private static void WalkTree()

{

**var tree = new EmployeeHierarchy();**

**WalkTree(tree);**

}

**private static void WalkTree(Tree<Employee> node)**

**{**

**}**

1. In the new overload of WalkTree, write a tree walking implementation that will first process an employee using a new instance of PayrollService (and calling GetPayrollDeduction). Then recursively call the method with the left node and right node. Include diagnostic information for when an employee starts and ends.

private static void WalkTree(Tree<Employee> node)

{

**if (node == null) return;**

**if (node.Data != null)**

**{**

**Employee employee = node.Data;**

**Console.WriteLine("Starting employee id {0}", employee.EmployeeID);**

**decimal span = new PayrollServices()**

**.GetPayrollDeduction(employee);**

**Console.WriteLine("Completed process for employee id {0} took {1}s",**

**employee.EmployeeID, span);**

**Console.WriteLine();**

**}**

**WalkTree(node.Left);**

**WalkTree(node.Right);**

}

1. **Run** the application and make note of the elapsed time, and the **order** in which the employees are processed.
2. Replace the two calls to WalkTree (for node.Left and node.Right) with a single call to **Parallel.Invoke**. Pass in lambda expressions to invoke WalkTree on the left and right nodes in parallel.

Parallel.Invoke(() => WalkTree(node.Left),

() => WalkTree(node.Right));

1. **Run** the application and make note of better elapsed time, and the different ordering in the processing.

# Part II – Task<T>

1. In the Main method, replace the call to WalkTree with a call to a new method: UseTask.

static void Main()

{

var watch = Stopwatch.StartNew();

**UseTask();**

Console.WriteLine("Total time: {0}s", watch.Elapsed.TotalSeconds);

}

private static void UseTask()

{

}

1. Inside UseTask, create a new instance of EmployeeList and PayrollServices.
2. Inside UseTask, create three tasks with Task.Factory.StartNew (task1, task2, task3). In the Action passed to start new, use the service to process a single employee (you can process just the first three employees in the list).

private static void UseTask()

{

var service = new PayrollServices();

var employees = new EmployeeList();

var task1 = Task.Factory.StartNew(() => service.GetPayrollDeduction(employees[0]));

var task2 = Task.Factory.StartNew(() => service.GetPayrollDeduction(employees[1]));

var task3 = Task.Factory.StartNew(() => service.GetPayrollDeduction(employees[2]));

}

1. **Run** the application.

Was the result surprising? If you are not sure why the program is behaving the way it does – ask the instructor – that’s why the instructor is here!

1. Inside UseTask, add a call to Wait on each task you created.

private static void UseTask()

{

var service = new PayrollServices();

var employees = new EmployeeList();

var task1 = Task.Factory.StartNew(() => service.GetPayrollDeduction(employees[0]));

var task2 = Task.Factory.StartNew(() => service.GetPayrollDeduction(employees[1]));

var task3 = Task.Factory.StartNew(() => service.GetPayrollDeduction(employees[2]));

**task1.Wait();**

**task2.Wait();**

**task3.Wait();**

}

1. **Run** the application. What’s different?
2. **Replace** the three calls to **Wait** in UseTask with a **single** call to **Task.WaitAll**.

Task.WaitAll(task1, task2, task3);

1. **Run** the application (the results should be the same as before, but just make sure they are ☺.
2. **Replace** the call to Task.WaitAll with some logic that will poll the tasks to see if they are completed (IsCompleted). Hint: you’ll need to setup a loop with a Thread.Sleep inside.

while(!task1.IsCompleted && !task2.IsCompleted && !task3.IsCompleted)

{

Thread.Sleep(1000);

}

1. **Run** the application. Once again, you should have roughly the same results.
2. Change the application to use WaitAll again.
3. Create a new Task – task4, that is a continuation (ContinueWith) of task1. Use the continuation to write a message to the console. Make sure to wait for the task to complete.

var task4 = task1.ContinueWith(t =>

{

Console.WriteLine("task4 !");

});

...

Task.WaitAll(task1, task2, task3, task4);

1. Now use the task4 continuation to write out the value returned from task1’s call to GetPayrollDeduction (hint: use the Result property of the antecedent task).

var task4 = task1.ContinueWith(t =>

{

Console.WriteLine(t.Result);

});

1. Run it and verify!

# Conclusion

Congratulations! You’ve now used one of the most beautifully designed APIs in all the .NET framework – the TPL API. With the TPL multithreading is easy and fun.