Thread Safety

Estimated time for completion: 60 minutes

Overview:

In this lab you explore the more interesting side of multithreaded programming, making it both safe and scalable.

Goals:

- Learn how to use monitors to prevent concurrent access to shared resources
- Learn how to use concurrent collections

Lab Notes:

Part 1: Dining Philosophers Problem

The dining philosopher's problem is summarized as five philosophers sitting at a table doing one of two things: eating or thinking. While eating, they are not thinking, and while thinking, they are not eating. The five philosophers sit at a circular table with a large bowl of spaghetti in the center. A fork is placed in between each pair of adjacent philosophers, and as such, each philosopher has one fork to his left and one fork to his right. As spaghetti is difficult to serve and eat with a single fork, it is assumed that a philosopher must eat with two forks. Each philosopher can only use the forks on his immediate left and immediate right (http://en.wikipedia.org/wiki/Dining_philosophers_problem)

1. Open the Before\ThreadSafety.sln solution, the active project will be **WinPhilosophers** this project provides a visualisation of the dining philosopher's problem; you will not be modifying any code in this project. The project that you will use for this part of the lab is the **DiningPhilosophers** project. Open up the project and you will see a **Fork**, Philosopher and DiningTable class. The Fork class simply models a fork. Each fork has an Id and an owner, the owner property is set when a philosopher picks up that fork. The **Philosopher** class models a philosopher sitting around the dining table. The **Run** method provides the main loop for the simulation simply cycling each philosopher through the states Thinking, Hungry and Eating. The **DiningTable** class simply aggregates all the philosopher and fork instances, and provides a single **Run** method to start the simulation. Examine all these classes and make sure you are happy with how they work. Notice that the **Fork** methods **PickUp** and **PutDown** have debug assertions to try and ensure that the object model stays in a valid state.
2. Run the solution, a window will appear showing the philosophers sitting around a dining table. Using the file menu select the **Start** option. This will call the **Run** method on the dining table, which in turn will call its philosophers **Run** method on a separate task. Resulting in each philosopher running in parallel. Each philosopher displays its state, and the number of bites of food they are making per second (they are a greedy bunch). As you will be aware there is no synchronization logic and it shouldn't be long before one of the defensive assertions in the **Fork** class fire.

3. The Assertions are firing because two philosophers are attempting to use the same fork, to solve this problem modify the philosopher's **Eat** method to lock the forks prior to picking them up. Hold onto the locks until the philosopher has put down the forks.
4. Re-run your code. It will now not throw any assertions…So all looks good.
5. Examine the Eat and Thinking methods inside the **Philospher** class you will notice that Thread.Sleep calls have been placed inside the methods, remove these sleep calls and re-run the app. It will deadlock very quickly.
6. Using the Debug menu, select the **Break All** option, then use the Parallel Stacks and Parallel Tasks debugging windows from the Visual Studio Debug menu to diagnose why the application has deadlocked.
7. The removal of the sleeps forced all the philosophers to aggressively grab forks, each philosopher grabs one fork and will not put it down until after they have eaten, and since there is not enough to eat they all starve. This problem can be fixed using deterministic locking order. Rather than always grab the left fork first each philosopher looks at the id of both forks and always grabs the fork with the lowest id first. Modify the constructor of the philosopher to assign the field **firstFork** to be the fork with the lowest id, and the **secondFork** to be assigned to the fork with the highest id.

```
firstFork = left.Id < right.Id ? left : right;
secondFork = left.Id < right.Id ? right : left;
```

8. Re-run the code, you will notice that it no longer deadlocks. The number of bites per second is rising for all philosophers thus fixing the deadlock problem.

Solution : After Part 2\ThreadSafety.sln

Part 2: Making it fair (Optional)

Whilst the previous solution worked, you may have noticed that one or two philosophers were managing to eat at a far greater pace than the others. Ultimately the locking strategy resulted in unfair scheduling. As a given philosopher gave up his forks, he was effectively picking them up straightaway resulting in other philosophers not having a chance to share the forks.

9. One way to make this fair is to limit the number of philosophers who can start to take forks at any one time. When more than half of the philosophers try and eat at any one time, someone will have to go hungry. One way of achieving this is to use a **Semaphore**. A semaphore is a synchronization primitive that unlike the monitor will allow many threads to acquire it at any one time up to a predefined limit. Once the limit has been reached all new threads trying to acquire will block waiting for a thread to release its acquisition. If a semaphore is created to the value of half the number of philosophers and if each philosopher acquires the semaphore before attempting to pick up the forks, and then release the semaphore once the forks have been picked up this will result in any philosopher who is eating to have to wait for hungry philosophers to eat before they can again eat.
10. Create a semaphore as part of the constructor for the **DiningTable** class. Initialise the semaphore to half the number of philosophers dining. Modify the constructor for the

philosophers class to receive this semaphore, and store it in a field called **tableSemaphore**.

```
SemaphoreSlim tableSemaphore = new SemaphoreSlim(nPhilosophers / 2);

for (int nPhilospher = 0; nPhilospher < philosophers.Length; nPhilospher++)
{
    philosophers[nPhilospher] = new Philosopher(nPhilospher,
   forks[nPhilospher], forks[(nPhilospher + 1) % nPhilosophers] ,
   tableSemaphore);
}
```

11. Modify the philosophers **Eat** method to acquire the semaphore before attempting to lock any forks. You acquire a semaphore by waiting on it, and then release it to relinquish

```
tableSemaphore.Wait();

lock (firstFork)
{
  lock (secondFork)
  {
     tableSemaphore.Release();
   // ...
  }
}
```

12. Re-run the application it will now be a lot fairer.

Solution : After Part 2\ThreadSafety.sln

Part 3: Concurrent Collections

In this part of the lab you will turn a single threaded web server into a multi-threaded web server utilizing the producer consumer pattern. The producer consumer pattern typically separates producers and consumers with a queue. In this case the web server will receive a request and place the request into a queue (producer) ready then to accept any new requests. A set of background tasks will monitor the queue, and when a task becomes free take an item from the queue (consumer) and process it.

Lab Notes:

If running on Windows Server 2008, Vista or Windows 7 you will need to first allow the webserver to host a site under a root url. Open an Administrator command prompt, and run the **netsh** command, and add an ACL for the web servers root URL and username of the current logged in user.

```
netsh
netsh>http add urlacl url=http://+:9001/Fractals user=DOMAIN\username

URL reservation successfully added
```

1. Set the **WebServer** project to be the active project. Run the solution, a console window will appear stating the server is up and running. If you get an exception it is most likely due to the URL ACL not being registered correctly.
2. Whilst keeping the server running open a browser and try out the following URL's

```
http://localhost:9001/Fractals/
http://localhost:9001/Fractals/time
```

3. You will have observed that it takes a reasonable amount of time to produce the fractal, and a very short period of time to display the time. If you request a fractal in one browser window, and immediately afterwards request the time in the other, you will see that you have to wait for the fractal to be generated before receiving the time, this is due to the web server being single threaded.
4. Open the **Program.cs** file inside the **WebServer** project, familiarise yourself with the code. The listener.GetContext block of code simply waits for a web request, on receipt of one returns a **HttpListenerContext** object that contains all the information necessary to handle the web request. Currently the request is being handled on the same thread that is waiting for new requests, and as such each request is being queued one behind each other. To solve this you will create a queue wrapped with a Blocking Collection to provide the basis for the implementation of the producer/consumer pattern. The queue will contain **HttpListenerContext** objects returned from the GetContext method.

```
private static BlockingCollection<HttpListenerContext> queue = new
   BlockingCollection<HttpListenerContext>( new
   ConcurrentQueue<HttpListenerContext>() );
```

5. Now create a consumer method, by adding a static method to the Program class called **RequestHandler**. This method will be responsible for getting available items from the queue and them processing them in the same way as the single threaded implementation. The difference being multiple tasks will be spawned using this method.

```
private static void RequestHandler()
{
 foreach (HttpListenerContext ctx in queue.GetConsumingEnumerable())
 {
     // . . .
 }
}
```

6. Now move the processing code from the main while loop into the **RequestHandler's** foreach loop, leaving the listener.GetContext method call behind.
7. Continue refactoring the main while loop to now simply add the result from **listener.GetContext** to the request queue.

```
while (true)
{
   HttpListenerContext ctx = listener.GetContext();

   queue.Add(ctx);
```

```
}
```

8. Finally before the while loop create and start five tasks with the tasks start method being **RequestHandler**.

```
for (int nTask = 0; nTask < 5; nTask++)
{
   Task.Factory.StartNew(RequestHandler);
}
```

9. Now rerun the application and verify now that time requests can be returned without waiting for previous requests to complete first.

Solution: After\ThreadSafety.sln