## 进程间同步通信

private Process \_process;  
 public delegate void OutputReceivedDelegate(string output);  
 public event OutputReceivedDelegate OutputReceived;

此段代码定义一个 ProcessSynchronousCommunication 的公共类，并且声明一个私有的 Process 类型的变量 \_process，用于管理和监控系统进程；定义一个名为 OutputReceivedDelegate 的委托，它接受一个字符串类型的参数并没有返回值，接着定义一个此类型的事件 OutputReceived，使外部可以订阅该事件并处理输出数据。

public ProcessSynchronousCommunication()  
 {  
 \_process = new Process();  
  
 \_process.StartInfo.FileName = "ping.exe";  
  
 \_process.StartInfo.UseShellExecute = false;  
 \_process.StartInfo.RedirectStandardOutput = true;   
 \_process.StartInfo.RedirectStandardInput = true;   
 \_process.StartInfo.CreateNoWindow = true;  
  
 \_process.OutputDataReceived += (sender, e) =>  
 {  
 if (e.Data != null)   
 {  
 OutputReceived?.Invoke(e.Data);   
 }  
 };  
 }

在 ProcessSynchronousCommunication 类的构造方法中，首先初始化了一个新的进程实例。为了运行系统的 ping 命令，进程的启动信息被配置为 "ping.exe"。进程的启动参数还被定制为：1) 不使用 shell 来执行命令，2) 重定向标准输出和输入流，以便于程序可以同步捕获命令的输出和向命令输入数据，3) 在启动进程时不创建新窗口，为了使命令执行的后台运行不打扰用户。此外，还为进程的输出数据添加了一个事件处理程序，该程序能够实时捕获命令的输出，并通过 OutputReceived 事件将其传达给外部订阅者。

public void ExecutePingSync(string ipAddress)  
 {  
 \_process.StartInfo.Arguments = ipAddress + " -n 10";  
 \_process.Start();  
  
 string line;  
 while ((line = \_process.StandardOutput.ReadLine()) != null)  
 {  
 OutputReceived?.Invoke(line);  
 }  
  
 \_process.WaitForExit();  
 \_process.Close();  
 }

ExecutePingSync 方法是 ProcessSynchronousCommunication 类的核心功能，它实现了对指定 IP 地址进行同步的 ping 操作。方法首先配置进程的启动参数以 ping 输入的 IP 地址 10 次，接着启动该进程。随后，方法同步地读取进程的标准输出流中的每一行数据，并通过已定义的 OutputReceived 事件实时地传达给外部订阅者。在数据完全读取后，方法等待进程完成其执行，并随后关闭进程，确保所有关联的资源被正确释放。

## 进程间异步通信

\_process.OutputDataReceived += (sender, e) =>  
 {  
 if (e.Data != null)  
 {  
 OutputReceived?.Invoke(e.Data);  
 }  
 };

在这段代码中，通过 \_process.OutputDataReceived 事件订阅了一个匿名的回调函数，目的是实现进程间的异步通信。OutputDataReceived 事件是 System.Diagnostics.Process 类的一个特性，用于异步读取进程的标准输出流。当进程通过标准输出发送数据时，这个事件被触发，并且事件的参数 e 中包含了输出的数据。具体来说，e.Data 属性包含了从输出流中读取的一行字符串。在这个回调函数内部，首先检查 e.Data 是否为 null，这是一个常见的做法以确保不对空数据进行处理；如果 e.Data 包含有效数据，接下来则触发了一个自定义的事件 OutputReceived，将进程的输出数据作为参数传递出去。这里使用的 ?.Invoke(e.Data) 语法是C#中的空值条件操作符，它用于确保只有在有至少一个订阅者订阅 OutputReceived 事件时才会调用 Invoke 方法。整体而言，这段代码利用了.NET Framework提供的事件机制，实现了一个灵活且高效的进程间异步通信模型，使得外部订阅者能够实时获取并处理进程的输出数据。

public void ExecutePingAsync(string ipAddress)  
 {  
 \_process.StartInfo.Arguments = ipAddress + " -n 10";  
 \_process.Start();  
  
 \_process.BeginOutputReadLine();  
 }

这段代码定义了一个方法 ExecutePingAsync，该方法接受一个 ipAddress 字符串作为参数。首先，方法设置进程的启动参数以异步 ping 给定的 IP 地址 10 次。随后，方法启动该进程并调用 BeginOutputReadLine，这使得进程开始异步读取输出数据，这是与同步方法最大的区别。

与同步版本的 ExecutePingSync 方法不同，ExecutePingAsync 方法中并没有采用循环结构来连续读取每一行输出数据。在同步方法中，程序会阻塞在循环结构内，逐一读取每行输出，直到所有数据读取完毕。相反，在这里，我们使用 BeginOutputReadLine 方法，该方法异步地启动进程的标准输出的读取操作。这意味着 ExecutePingAsync 方法在启动读取操作后会立即返回，而不会等待读取操作完成。因此，程序的执行流可以继续进行，而不会被阻塞在这个方法内，增强了程序的响应性。当输出数据准备好时，由于我们已经为 \_process.OutputDataReceived 事件注册了回调，该回调会被触发并处理输出数据，实现了异步的数据处理模式。

### **综合分析**

在本次实验中，我们探讨了两种不同的进程通信方式：同步通信和异步通信。通过实际的操作和实验，我们对这两种机制的工作原理有了更加深入的了解。

在同步通信部分，首先，我们采用了命令行工具cmd来获取电脑的IP地址。然后，我们配置了进程以执行ping命令，尝试ping该地址十次。关键在于我们使用了重定向技术，将进程的标准输入和输出与我们的应用程序关联，从而能够在不打开新窗口的情况下从进程捕获输出数据。同步通信的特点是，我们必须等待进程完成其执行（在这种情况下，是完成所有的ping操作）之后，才能收到并显示完整的输出。这意味着在进程执行期间，我们的程序处于等待状态，直到接收到完整的数据。

另一方面，异步通信采用了与同步通信类似的重定向技术，但它的工作方式完全不同。在异步通信中，我们为进程的输出设置了一个回调函数。这意味着每当进程有新的输出数据时，这个回调函数都会被立即触发，无需等待进程完成。这提供了一种非常高效的方式来处理数据，因为我们可以实时获取和显示输出，而不是在进程完成后一次性显示。这对于那些需要实时反馈的应用程序，如网络监控工具，是非常有用的。

通过这个实验，我们了解到同步和异步通信之间的根本区别在于它们如何处理输出数据。同步通信要求我们等待整个进程完成，然后一次性处理所有的输出，而异步通信则允许我们实时地处理输出，每当有新的数据到达时。

总的来说，本次实验成功地展示了同步和异步进程通信的工作原理和应用场景，并通过实际的代码实现加深了我们对这两种机制的理解。这为我们今后开发需要进程间通信的应用程序提供了宝贵的经验和知识。