IPC（Inter-Process Communication，进程间通信）

应用程序之间的数据交换（互相通讯）一直是困扰广大程序员的难题，尽管已经出现了各式各样的解决方案，但迄今为止没有哪一种方案是完美无缺的。因此，只有学习并了解了它们的优缺点后，才能在特定的情况下选择最佳方案，以满足最终的要求。

　进程是装入内存并准备执行的程序，每个进程都有私有的虚拟地址空间，由代码、数据以及它可利用的系统资源(如文件、管道等)组成。多进程/多线 程是Windows操作系统的一个基本特征。Microsoft Win32应用编程接口(Application Programming Interface, API)提供了大量支持应用程序间数据共享和交换的机制，这些机制行使的活动称为进程间通信(InterProcess Communication, IPC)，进程通信就是指不同进程间进行数据共享和数据交换。  
　　正因为使用Win32 API进行进程通信方式有多种，如何选择恰当的通信方式就成为应用开发中的一个重要问题，下面本文将对Win32中进程通信的几种方法加以分析和比较。

**2 进程通信方法**

**2.1 文件映射**  
　　文件映射(Memory-Mapped Files)能使进程把文件内容当作进程地址区间一块内存那样来对待。因此，进程不必使用文件I/O操作，只需简单的指针操作就可读取和修改文件的内容。  
　　Win32 API允许多个进程访问同一文件映射对象，各个进程在它自己的地址空间里接收内存的指针。通过使用这些指针，不同进程就可以读或修改文件的内容，实现了对文件中数据的共享。  
　　应用程序有三种方法来使多个进程共享一个文件映射对象。  
　　(1)继承：第一个进程建立文件映射对象，它的子进程继承该对象的句柄。  
　　(2)命名文件映射：第一个进程在建立文件映射对象时可以给该对象指定一个名字(可与文件名不同)。第二个进程可通过这个名字打开此文件映射对象。另外，第一个进程也可以通过一些其它IPC机制(有名管道、邮件槽等)把名字传给第二个进程。  
　　(3)句柄复制：第一个进程建立文件映射对象，然后通过其它IPC机制(有名管道、邮件槽等)把对象句柄传递给第二个进程。第二个进程复制该句柄就取得对该文件映射对象的访问权限。  
　　文件映射是在多个进程间共享数据的非常有效方法，有较好的安全性。但文件映射只能用于本地机器的进程之间，不能用于网络中，而开发者还必须控制进程间的同步。

**2.2 共享内存**  
　　Win32 API中共享内存(Shared Memory)实际就是文件映射的一种特殊情况。进程在创建文件映射对象时用0xFFFFFFFF来代替文件句柄(HANDLE)，就表示了对应的文件映 射对象是从操作系统页面文件访问内存，其它进程打开该文件映射对象就可以访问该内存块。由于共享内存是用文件映射实现的，所以它也有较好的安全性，也只能 运行于同一计算机上的进程之间。

**2.3 匿名管道**  
　　管道(Pipe)是一种具有两个端点的通信通道：有一端句柄的进程可以和有另一端句柄的进程通信。管道可以是单向－一端是只读的，另一端点是只写的；也可以是双向的一管道的两端点既可读也可写。  
　　匿名管道(Anonymous Pipe)是在父进程和子进程之间，或同一父进程的两个子进程之间传输数据的无名字的单向管道。通常由父进程创建管道，然后由要通信的子进程继承通道的读 端点句柄或写端点句柄，然后实现通信。父进程还可以建立两个或更多个继承匿名管道读和写句柄的子进程。这些子进程可以使用管道直接通信，不需要通过父进 程。  
　　匿名管道是单机上实现子进程标准I/O重定向的有效方法，它不能在网上使用，也不能用于两个不相关的进程之间。

**2.4 命名管道**  
　　命名管道(Named Pipe)是服务器进程和一个或多个客户进程之间通信的单向或双向管道。不同于匿名管道的是命名管道可以在不相关的进程之间和不同计算机之间使用，服务器 建立命名管道时给它指定一个名字，任何进程都可以通过该名字打开管道的另一端，根据给定的权限和服务器进程通信。  
　　命名管道提供了相对简单的编程接口，使通过网络传输数据并不比同一计算机上两进程之间通信更困难，不过如果要同时和多个进程通信它就力不从心了。  
**2.5 邮件槽**  
　　邮件槽(Mailslots)提供进程间单向通信能力，任何进程都能建立邮件槽成为邮件槽服务器。其它进程，称为邮件槽客户，可以通过邮件槽的名字给 邮件槽服务器进程发送消息。进来的消息一直放在邮件槽中，直到服务器进程读取它为止。一个进程既可以是邮件槽服务器也可以是邮件槽客户，因此可建立多个邮 件槽实现进程间的双向通信。  
　　通过邮件槽可以给本地计算机上的邮件槽、其它计算机上的邮件槽或指定网络区域中所有计算机上有同样名字的邮件槽发送消息。广播通信的消息长度不能超过400字节，非广播消息的长度则受邮件槽服务器指定的最大消息长度的限制。  
　　邮件槽与命名管道相似，不过它传输数据是通过不可靠的数据报(如TCP/IP协议中的UDP包)完成的，一旦网络发生错误则无法保证消息正确地接收， 而命名管道传输数据则是建立在可靠连接基础上的。不过邮件槽有简化的编程接口和给指定网络区域内的所有计算机广播消息的能力，所以邮件槽不失为应用程序发 送和接收消息的另一种选择。

**2.6 剪贴板**  
　　剪贴板(Clipped Board)实质是Win32 API中一组用来传输数据的函数和消息，为Windows应用程序之间进行数据共享提供了一个中介，Windows已建立的剪切(复制)－粘贴的机制为不 同应用程序之间共享不同格式数据提供了一条捷径。当用户在应用程序中执行剪切或复制操作时，应用程序把选取的数据用一种或多种格式放在剪贴板上。然后任何 其它应用程序都可以从剪贴板上拾取数据，从给定格式中选择适合自己的格式。  
　　剪贴板是一个非常松散的交换媒介，可以支持任何数据格式，每一格式由一无符号整数标识，对标准(预定义)剪贴板格式，该值是Win32 API定义的常量；对非标准格式可以使用Register Clipboard Format函数注册为新的剪贴板格式。利用剪贴板进行交换的数据只需在数据格式上一致或都可以转化为某种格式就行。但剪贴板只能在基于Windows的 程序中使用，不能在网络上使用。

**2.7 动态数据交换**  
　　动态数据交换(DDE)是使用共享内存在应用程序之间进行数据交换的一种进程间通信形式。应用程序可以使用DDE进行一次性数据传输，也可以当出现新数据时，通过发送更新值在应用程序间动态交换数据。  
　　DDE和剪贴板一样既支持标准数据格式(如文本、位图等)，又可以支持自己定义的数据格式。但它们的数据传输机制却不同，一个明显区别是剪贴板操作几 乎总是用作对用户指定操作的一次性应答－如从菜单中选择Paste命令。尽管DDE也可以由用户启动，但它继续发挥作用一般不必用户进一步干预。DDE有 三种数据交换方式：  
　　(1) 冷链：数据交换是一次性数据传输，与剪贴板相同。  
　　(2) 温链：当数据交换时服务器通知客户，然后客户必须请求新的数据。  
　　(3) 热链：当数据交换时服务器自动给客户发送数据。  
　　DDE交换可以发生在单机或网络中不同计算机的应用程序之间。开发者还可以定义定制的DDE数据格式进行应用程序之间特别目的IPC，它们有更紧密耦合的通信要求。大多数基于Windows的应用程序都支持DDE。

**2.8 对象连接与嵌入**  
　　应用程序利用对象连接与嵌入(OLE)技术管理复合文档(由多种数据格式组成的文档)，OLE提供使某应用程序更容易调用其它应用程序进行数据编辑的 服务。例如，OLE支持的字处理器可以嵌套电子表格，当用户要编辑电子表格时OLE库可自动启动电子表格编辑器。当用户退出电子表格编辑器时，该表格已在 原始字处理器文档中得到更新。在这里电子表格编辑器变成了字处理器的扩展，而如果使用DDE，用户要显式地启动电子表格编辑器。  
　　同DDE技术相同，大多数基于Windows的应用程序都支持OLE技术。

**2.9 动态连接库**  
　　Win32动态连接库(DLL)中的全局数据可以被调用DLL的所有进程共享，这就又给进程间通信开辟了一条新的途径，当然访问时要注意同步问题。  
　　虽然可以通过DLL进行进程间数据共享，但从数据安全的角度考虑，我们并不提倡这种方法，使用带有访问权限控制的共享内存的方法更好一些。

**2.10 远程过程调用**  
　　Win32 API提供的远程过程调用(RPC)使应用程序可以使用远程调用函数，这使在网络上用RPC进行进程通信就像函数调用那样简单。RPC既可以在单机不同进程间使用也可以在网络中使用。  
　　由于Win32 API提供的RPC服从OSF-DCE(Open Software Foundation Distributed Computing Environment)标准。所以通过Win32 API编写的RPC应用程序能与其它操作系统上支持DEC的RPC应用程序通信。使用RPC开发者可以建立高性能、紧密耦合的分布式应用程序。

**2.11 NetBios函数**  
　　Win32 API提供NetBios函数用于处理低级网络控制，这主要是为IBM NetBios系统编写与Windows的接口。除非那些有特殊低级网络功能要求的应用程序，其它应用程序最好不要使用NetBios函数来进行进程间通信。

**2.12 Sockets**  
　　Windows Sockets规范是以U.C.Berkeley大学BSD UNIX中流行的Socket接口为范例定义的一套Windows下的网络编程接口。除了Berkeley Socket原有的库函数以外，还扩展了一组针对Windows的函数，使程序员可以充分利用Windows的消息机制进行编程。  
　　现在通过Sockets实现进程通信的网络应用越来越多，这主要的原因是Sockets的跨平台性要比其它IPC机制好得多，另外WinSock 2.0不仅支持TCP/IP协议，而且还支持其它协议(如IPX)。Sockets的唯一缺点是它支持的是底层通信操作，这使得在单机的进程间进行简单数 据传递不太方便，这时使用下面将介绍的WM\_COPYDATA消息将更合适些。

**2.13 WM\_COPYDATA消息**  
　　WM\_COPYDATA是一种非常强大却鲜为人知的消息。当一个应用向另一个应用传送数据时，发送方只需使用调用SendMessage函数，参数是 目的窗口的句柄、传递数据的起始地址、WM\_COPYDATA消息。接收方只需像处理其它消息那样处理WM\_COPY DATA消息，这样收发双方就实现了数据共享。  
　　WM\_COPYDATA是一种非常简单的方法，它在底层实际上是通过文件映射来实现的。它的缺点是灵活性不高，并且它只能用于Windows平台的单机环境下。

**3 结束语**

　　Win32 API为应用程序实现进程间通信提供了如此多种选择方案，那么开发者如何进行选择呢？通常在决定使用哪种IPC方法之前应考虑下一些问题，如应用程序是在网络环境下还是在单机环境下工作等。

**内存磁盘**

RAMdisk的方法

VSuite RAMDisk Server

内存磁盘制作工具。需内ramdisk,采用windows自带的是附加磁盘 ，在其他磁盘上创建一个文件。需要采用专业的虚拟内存制作工作。

Crasydisk

内存读写速度测试工具。

**共享内存方法：**

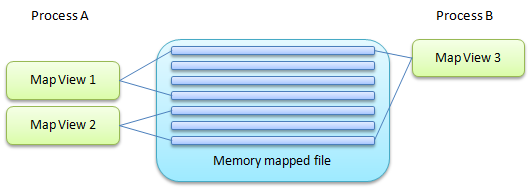
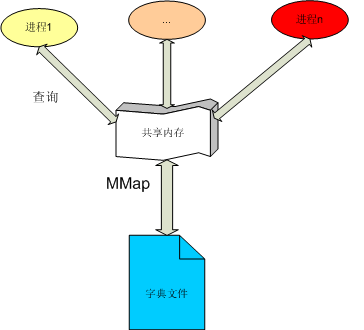
http://www.cnblogs.com/zeroone/archive/2012/04/18/2454776.html

C# .Net 多进程同步 通信 共享内存 内存映射文件 Memory Mapped 转

节点通信存在两种模型：共享内存（Shared memory）和消息传递（Messages passing）。

        内存映射文件对于托管世界的开发人员来说似乎很陌生，但它确实已经是很远古的技术了，而且在操作系统中地位相当。实际上，任何想要共享数据的通信模型都会在幕后使用它。

**内存映射文件究竟是个什么**？内存映射文件允许你保留一块地址空间，然后将该物理存储映射到这块内存空间中进行操作。物理存储是文件管理，而内存映射文件是**操作系统级内存管理**。

****

**优势**：  
     1.访问磁盘文件上的数据不需执行I/O操作和缓存操作(当访问文件数据时，作用尤其显著)；  
     2.让运行在同一台机器上的多个进程共享数据(单机多进程间数据通信效率最高)；

       利用文件与内存空间之间的映射，应用程序（包括多个进程）可以通过直接在内存中进行读写来修改文件。.NET Framework 4 用托管代码按照本机Windows函数访问内存映射文件的方式来访问内存映射文件，[管理 Win32 中的内存映射文件](http://go.microsoft.com/fwlink/?linkid=180801) 。

有两种类型的内存映射文件：

* 持久内存映射文件

持久文件是与磁盘上的源文件关联的内存映射文件。在最后一个进程使用完此文件后，数据将保存到磁盘上的源文件中。这些内存映射文件适合用来处理非常大的源文件。

* 非持久内存映射文件

非持久文件是未与磁盘上的源文件关联的内存映射文件。当最后一个进程使用完此文件后，数据将丢失，并且垃圾回收功能将回收此文件。这些文件适用于为**进程间通信** (IPC) 创建共享内存。

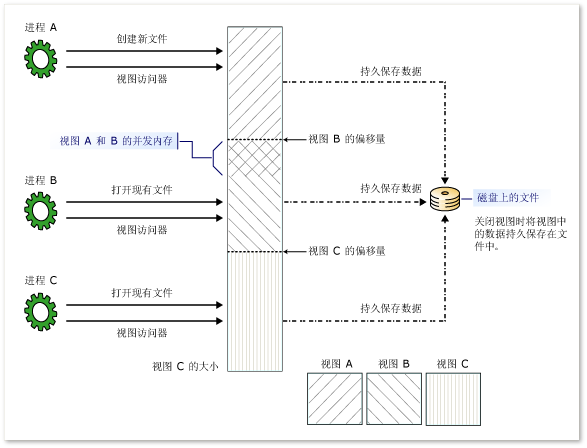
    1）在多个进程之间进行共享（进程可通过使用由创建同一内存映射文件的进程所指派的**公用名**来映射到此文件）。

    2）若要使用一个内存映射文件，则必须**创建该内存映射文件的完整视图或部分视图**。还可以创建内存映射文件的同一部分的多个视图，进而创建**并发内存**。**为了使两个视图能够并发，必须基于同一内存映射文件创建这两个视图**。

    3）如果**文件大于应用程序用于内存映射的逻辑内存空间**（在 32 位计算机上为**2GB**），则还需要使用多个视图。

有两种类型的视图：流访问视图和随机访问视图。使用流访问视图可对文件进行顺序访问；在使用持久文件时，随机访问视图是首选方法。

**.Net 共享内存 内存映射文件原理**：通过**操作系统的内存管理器**访问的，因此会自动将此文件分隔为多个页，并根据需要对其进行访问。您不需要自行处理内存管理。如下图：



**C# .Net 共享内存 演示代码**如下：

    //持久内存映射文件:基于现有文件创建一个具有指定公用名的内存映射文件

    using (var mmf = MemoryMappedFile.CreateFromFile(@"c:\内存映射文件.data", FileMode.Open, "公用名"))  
    {  
        //通过指定的 偏移量和大小 创建**内存映射文件视图服务器**  
        using (var accessor = mmf.CreateViewAccessor(offset, length)) //偏移量，可以控制数据存储的内存位置；大小，用来控制存储所占用的空间  
        {  
            //Marshal提供了一个方法集，这些方法用于分配非托管内存、复制非托管内存块、将托管类型转换为非托管类型，此外还提供了在与非托管代码交互时使用的其他杂项方法。

            int size = Marshal.SizeOf(typeof(char));

            //修改内存映射文件视图  
            for (long i = 0; i < length; i += size)  
            {  
                char c= accessor.ReadChar(i);  
                accessor.Write(i, ref c);  
            }  
        }  
    }

    //另一个进程或线程可以，在系统内存中打开一个具有指定名称的现有内存映射文件

    using (var mmf = MemoryMappedFile.OpenExisting("公用名"))  
    {  
        using (var accessor = mmf.CreateViewAccessor(4000000, 2000000))  
        {  
            int size = Marshal.SizeOf(typeof(char));  
            for (long i = 0; i < length; i += size)  
            {  
                char c = accessor.ReadChar(i);  
                accessor.Write(i, ref c);  
            }  
        }  
    }

    //非持久内存映射文件：未映射到磁盘上的现有文件的内存映射文件

    using (MemoryMappedFile mmf = MemoryMappedFile.CreateNew("testmap", 10000))  
    {  
        bool mutexCreated;  
        //进程间同步  
        Mutex mutex = newMutex(true, "testmapmutex", out mutexCreated);  
        using (var stream = mmf.CreateViewStream()) //创建文件内存视图流 基于流的操作  
        {  
            var writer = newBinaryWriter(stream);  
            writer.Write(1);  
        }  
        mutex.ReleaseMutex();

        Console.WriteLine("Start Process B and press ENTER to continue.");  
        Console.ReadLine();

        mutex.WaitOne();  
        using (MemoryMappedViewStream stream = mmf.CreateViewStream())  
        {  
            var reader = newBinaryReader(stream);  
            Console.WriteLine("Process A says: {0}", reader.ReadBoolean());  
            Console.WriteLine("Process B says: {0}", reader.ReadBoolean());  
        }  
        mutex.ReleaseMutex();  
    }

    using (MemoryMappedFile mmf = MemoryMappedFile.OpenExisting("testmap"))  
    {  
         Mutex mutex = Mutex.OpenExisting("testmapmutex");  
        mutex.WaitOne();  
       using (var stream = mmf.CreateViewStream(1, 0))//注意这里的偏移量  
        {  
            var writer = newBinaryWriter(stream);  
            writer.Write(0);  
        }  
        mutex.ReleaseMutex();

    }

**C# .Net  进程间通信 共享内存** 完整示例： C#共享内存非持久化方式通讯的例子，通讯时的线程和进程控制也没有问题。如下是实现的代码。

先启动消息服务IMServer\_Message，

再启动状态服务IMServer\_State，

IMServer\_Message回车一次(创建共享内存公用名和公用线程锁，并视图流方式写共享内存),

IMServer\_State回车一次（获取共享内存并视图流方式写、视图访问器写入结构体类型）

并立刻IMServer\_Message再回车一次（读取刚刚写入的信息），

观察IMServer\_State屏显变化并等待(线程锁)约5s（线程锁被释放）后

在IMServer\_Message上观察屏显（显示刚刚写入共享内存的信息）

**IMServer\_Message.exe 代码**

using System;  
using System.IO;  
using System.IO.MemoryMappedFiles;  
using System.Runtime.InteropServices;  
using System.Threading;

namespace IMServer\_Message  
{  
    /// <summary>  
    /// 用于共享内存方式通信的 值类型 结构体  
    /// </summary>  
    public struct ServiceMsg  
    {  
        public int Id;  
        public long NowTime;  
    }

    internal class Program  
    {  
        private static void Main(string[] args)  
        {  
            Console.Write("请输入共享内存公用名(默认:testmap):");  
            string shareName = Console.ReadLine();  
            if (string.IsNullOrEmpty(shareName))  
                shareName = "testmap";  
            using (MemoryMappedFile mmf = MemoryMappedFile.CreateOrOpen(shareName, 1024000,MemoryMappedFileAccess.ReadWrite))  
            {  
                bool mutexCreated;  
                //进程间同步  
                var mutex = new Mutex(true, "testmapmutex", out mutexCreated);  
                using (MemoryMappedViewStream stream = mmf.CreateViewStream()) //创建文件内存视图流  
                {  
                    var writer = new BinaryWriter(stream);  
                    for (int i = 0; i < 5; i++)  
                    {  
                        writer.Write(i);  
                        Console.WriteLine("{0}位置写入流:{0}", i);  
                    }  
                }

                mutex.ReleaseMutex();

                Console.WriteLine("启动状态服务，按【回车】读取共享内存数据");  
                Console.ReadLine();

                mutex.WaitOne();  
                using (MemoryMappedViewStream stream = mmf.CreateViewStream())  
                {  
                    var reader = new BinaryReader(stream);  
                    for (int i = 0; i < 10; i++)  
                    {  
                        Console.WriteLine("{1}位置:{0}", reader.ReadInt32(), i);  
                    }  
                }

                using (MemoryMappedViewAccessor accessor = mmf.CreateViewAccessor(1024, 10240))  
                {  
                    int colorSize = Marshal.SizeOf(typeof (ServiceMsg));  
                    ServiceMsg color;  
                    for (int i = 0; i < 50; i += colorSize)  
                    {  
                        accessor.Read(i, out color);  
                        Console.WriteLine("{1}\tNowTime:{0}", new DateTime(color.NowTime), color.Id);  
                    }  
                }  
                mutex.ReleaseMutex();  
            }  
            Console.WriteLine("测试： 我是 即时通讯 - 消息服务 我启动啦！！！");  
            Console.ReadKey();  
        }  
    }  
}

**IMServer\_State.exe代码**

using System;  
using System.IO;  
using System.IO.MemoryMappedFiles;  
using System.Runtime.InteropServices;  
using System.Threading;

namespace IMServer\_State  
{  
    /// <summary>  
    /// 用于共享内存方式通信的 值类型 结构体  
    /// </summary>  
    public struct ServiceMsg  
    {  
        public int Id;  
        public long NowTime;  
    }

    internal class Program  
    {  
        private static void Main(string[] args)  
        {  
            Console.Write("请输入共享内存公用名(默认:testmap):");  
            string shareName = Console.ReadLine();  
            if (string.IsNullOrEmpty(shareName))  
                shareName = "testmap";  
            using (MemoryMappedFile mmf = MemoryMappedFile.CreateOrOpen(shareName, 1024000,MemoryMappedFileAccess.ReadWrite))  
            {  
                Mutex mutex = Mutex.OpenExisting("testmapmutex");  
                mutex.WaitOne();  
                using (MemoryMappedViewStream stream = mmf.CreateViewStream(20, 0)) //注意这里的偏移量  
                {  
                    var writer = new BinaryWriter(stream);  
                    for (int i = 5; i < 10; i++)  
                    {  
                        writer.Write(i);  
                        Console.WriteLine("{0}位置写入流:{0}", i);  
                    }  
                }  
                using (MemoryMappedViewAccessor accessor = mmf.CreateViewAccessor(1024, 10240))  
                {  
                    int colorSize = Marshal.SizeOf(typeof (ServiceMsg));  
                    var color = new ServiceMsg();  
                    for (int i = 0; i < colorSize\*5; i += colorSize)  
                    {  
                        color.Id = i;  
                        color.NowTime = DateTime.Now.Ticks;  
                        //accessor.Read(i, out color);  
                        accessor.Write(i, ref color);  
                        Console.WriteLine("{1}\tNowTime:{0}", new DateTime(color.NowTime), color.Id);  
                        Thread.Sleep(1000);  
                    }  
                }  
                Thread.Sleep(5000);

                mutex.ReleaseMutex();  
            }  
            Console.WriteLine("测试： 我是 即时通讯 - 状态服务 我启动啦！！！");  
            Console.ReadKey();  
        }  
    }  
}

进程间的互斥与同步

<http://www.soaspx.com/dotnet/csharp/csharp_20120415_8950.html>

### 简介

    进程同步是一个操作系统级别的概念,是在多道程序的环境下，存在着不同的制约关系，为了协调这种互相制约的关系，实现资源共享和进程协作，从而避免进程之间的冲突，引入了进程同步。

### 临界资源

    在操作系统中，进程是占有资源的最小单位（线程可以访问其所在进程内的所有资源，但线程本身并不占有资源或仅仅占有一点必须资源）。但对于某些资源来说，其在同一时间只能被一个进程所占用。这些一次只能被一个进程所占用的资源就是所谓的临界资源。典型的临界资源比如物理上的打印机，或是存在硬盘或内存中被多个进程所共享的一些变量和数据等(如果这类资源不被看成临界资源加以保护，那么很有可能造成丢数据的问题)。

    对于临界资源的访问，必须是互诉进行。也就是当临界资源被占用时，另一个申请临界资源的进程会被阻塞，直到其所申请的临界资源被释放。而进程内访问临界资源的代码被成为临界区。

    对于临界区的访问过程分为四个部分：

    1.进入区:查看临界区是否可访问，如果可以访问，则转到步骤二，否则进程会被阻塞

    2.临界区:在临界区做操作

    3.退出区:清除临界区被占用的标志

    4.剩余区：进程与临界区不相关部分的代码

### 进程间同步和互诉的概念

**进程同步**

    进程同步也是进程之间直接的制约关系，是为完成某种任务而建立的两个或多个线程，这个线程需要在某些位置上协调他们的工作次序而等待、传递信息所产生的制约关系。进程间的直接制约关系来源于他们之间的合作。

    比如说进程A需要从缓冲区读取进程B产生的信息，当缓冲区为空时，进程B因为读取不到信息而被阻塞。而当进程A产生信息放入缓冲区时，进程B才会被唤醒。概念如图1所示。

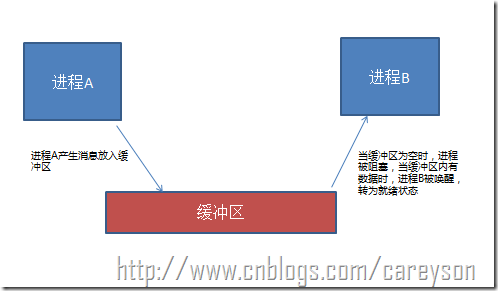
[](http://images.cnblogs.com/cnblogs_com/CareySon/201204/201204141746127533.png)

     图1.进程之间的同步

     用C#代码模拟进程之间的同步如代码1所示。

class ProcessSyn

{

private static Mutex mut = new Mutex();

static void Main()

{

Console.WriteLine("进程1执行完了进程2才能执行.......");

Thread Thread1 = new Thread(new ThreadStart([Proc](http://search.microsoft.com/default.asp?so=RECCNT&siteid=us%2Fdev&p=1&nq=NEW&qu=Proc&IntlSearch=&boolean=PHRASE&ig=01&i=09&i=99)1));

Thread Thread2 = new Thread(new ThreadStart([Proc](http://search.microsoft.com/default.asp?so=RECCNT&siteid=us%2Fdev&p=1&nq=NEW&qu=Proc&IntlSearch=&boolean=PHRASE&ig=01&i=09&i=99)2));

Thread1.Start();

Thread2.Start();

Console.ReadKey();

}

private static void [Proc](http://search.microsoft.com/default.asp?so=RECCNT&siteid=us%2Fdev&p=1&nq=NEW&qu=Proc&IntlSearch=&boolean=PHRASE&ig=01&i=09&i=99)1()

{

mut.WaitOne();

Console.WriteLine("线程1执行操作....");

Thread.Sleep(3000);

mut.ReleaseMutex();//V操作

}

private static void [Proc](http://search.microsoft.com/default.asp?so=RECCNT&siteid=us%2Fdev&p=1&nq=NEW&qu=Proc&IntlSearch=&boolean=PHRASE&ig=01&i=09&i=99)2()

{

mut.WaitOne();//P操作

Console.WriteLine("线程2执行操作....");

mut.WaitOne();

}

}

    代码1.C#模拟进程之间的同步

    运行结果如图2所示。

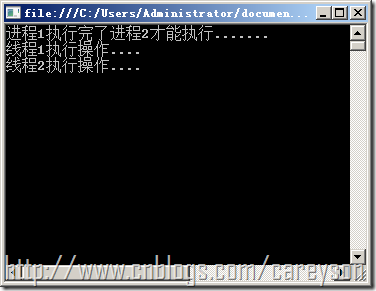
[](http://images.cnblogs.com/cnblogs_com/CareySon/201204/201204141746517483.png)

    图2.运行结果

**进程互斥**

    进程互斥是进程之间的间接制约关系。当一个进程进入临界区使用临界资源时，另一个进程必须等待。只有当使用临界资源的进程退出临界区后，这个进程才会解除阻塞状态。

    比如进程B需要访问打印机，但此时进程A占有了打印机，进程B会被阻塞，直到进程A释放了打印机资源,进程B才可以继续执行。概念如图3所示。

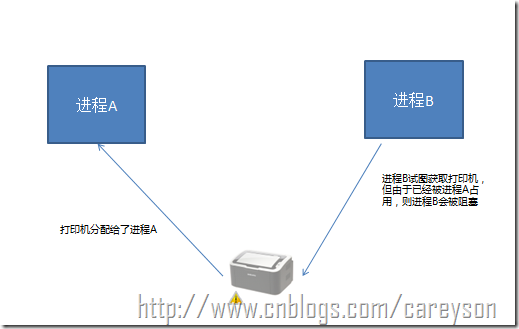
[](http://images.cnblogs.com/cnblogs_com/CareySon/201204/201204141747167616.png)

     图3.进程之间的互斥

     用C#模拟进程之间的互斥，这里我启动了5个线程，但同一时间内只有一个线程能对临界资源进行访问。如代码2所示。

class ProcessMutex

{

private static Mutex mut = new Mutex();

private const int numThreads = 5;

static void Main()

{

for (int i = 0; i <= numThreads; i++)

{

Thread myThread = new Thread(new ThreadStart(UseResource));

myThread.Name = String.Format("线程{0}", i + 1);

myThread.Start();

}

Console.ReadKey();

}

//同步

private static void UseResource()

{

// 相当于P操作

mut.WaitOne();

/\*下面代码是线程真正的工作\*/

Console.WriteLine("{0}已经进入临界区",

Thread.CurrentThread.Name);

Random r = new Random();

int rNum = r.Next(2000);

Console.WriteLine("{0}执行操作，执行时间为{1}ms", Thread.CurrentThread.Name,rNum);

Thread.Sleep(rNum);

Console.WriteLine("{0}已经离开临界区\r\n",

Thread.CurrentThread.Name);

/\*线程工作结束\*/

// 相当于V操作

mut.ReleaseMutex();

}

//互斥

}

    代码2.C#模拟进程之间的互斥

     运行结果如图4所示。

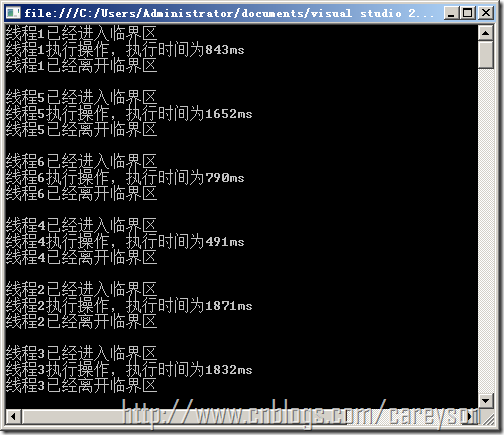
[](http://images.cnblogs.com/cnblogs_com/CareySon/201204/201204141747587090.png)

    图4.C#模拟进程互斥

### 实现临界区互斥的基本方法

**硬件实现方法**

    通过硬件实现临界区最简单的办法就是关CPU的中断。从计算机原理我们知道，CPU进行进程切换是需要通过中断来进行。如果屏蔽了中断那么就可以保证当前 进程顺利的将临界区代码执行完，从而实现了互斥。这个办法的步骤就是:屏蔽中断--执行临界区--开中断。但这样做并不好，这大大限制了处理器交替执行任 务的能力。并且将关中断的权限交给用户代码，那么如果用户代码屏蔽了中断后不再开，那系统岂不是跪了？

    还有硬件的指令实现方式，这个方式和接下来要说的信号量方式如出一辙。但是通过硬件来实现，这里就不细说了。

**信号量实现方式**

这也是我们比较熟悉P V操作。通过设置一个表示资源个数的信号量S，通过对信号量S的P和V操作来实现进程的的互斥。

    P和V操作分别来自荷兰语Passeren和Vrijgeven，分别表示占有和释放。P V操作是操作系统的原语，意味着具有原子性。

    P操作首先减少信号量，表示有一个进程将占用或等待资源，然后检测S是否小于0,如果小于0则阻塞，如果大于0则占有资源进行执行。

    V操作是和P操作相反的操作，首先增加信号量，表示占用或等待资源的进程减少了1个。然后检测S是否小于0，如果小于0则唤醒等待使用S资源的其它进程。

    前面我们C#模拟进程的同步和互斥其实算是信号量进行实现的。

### 一些经典利用信号量实现同步的问题

**生产者--消费者问题**

    问题描述:生产者-消费者问题是一个经典的进程同步问题，该问题最早由Dijkstra提出，用以演示他提出的信号量机制。本作业要求设计在同一个进程地 址空间内执行的两个线程。生产者线程生产物品，然后将物品放置在一个空缓冲区中供消费者线程消费。消费者线程从缓冲区中获得物品，然后释放缓冲区。当生产 者线程生产物品时，如果没有空缓冲区可用，那么生产者线程必须等待消费者线程释放出一个空缓冲区。当消费者线程消费物品时，如果没有满的缓冲区，那么消费 者线程将被阻塞，直到新的物品被生产出来

    这里生产者和消费者是既同步又互斥的关系，首先只有生产者生产了，消费着才能消费，这里是同步的关系。但他们对于临界区的访问又是互斥的关系。因此需要三个信号量empty和full用于同步缓冲区，而mut变量用于在访问缓冲区时是互斥的。

    利用C#模拟生产者和消费者的关系如代码3所示。

class ProducerAndCustomer

{

//临界区信号量

private static Mutex mut = new Mutex();

private static Semaphore empty = new Semaphore(5, 5);//空闲缓冲区

private static Semaphore full = new Semaphore(0, 5);

//生产者-消费者模拟

static void Main()

{

Console.WriteLine("生产者消费者模拟......");

for (int i = 1; i < 9; i++)

{

Thread Thread1 = new Thread(new ThreadStart(Producer));

Thread Thread2 = new Thread(new ThreadStart(Customer));

Thread1.Name = String.Format("生产者线程{0}", i);

Thread2.Name = String.Format("消费者线程{0}", i);

Thread1.Start();

Thread2.Start();

}

Console.ReadKey();

}

private static void Producer()

{

Console.WriteLine("{0}已经启动",Thread.CurrentThread.Name);

empty.WaitOne();//对empty进行P操作

mut.WaitOne();//对mut进行P操作

Console.WriteLine("{0}放入数据到临界区", Thread.CurrentThread.Name);

Thread.Sleep(1000);

mut.ReleaseMutex();//对mut进行V操作

full.Release();//对full进行V操作

}

private static void Customer()

{

Console.WriteLine("{0}已经启动", Thread.CurrentThread.Name);

Thread.Sleep(12000);

full.WaitOne();//对full进行P操作

mut.WaitOne();//对mut进行P操作

Console.WriteLine("{0}读取临界区", Thread.CurrentThread.Name);

mut.ReleaseMutex();//对mut进行V操作

empty.Release();//对empty进行V操作

}

}

   代码3.使用C#模拟生产者和消费者的关系

    运行结果如图5所示。

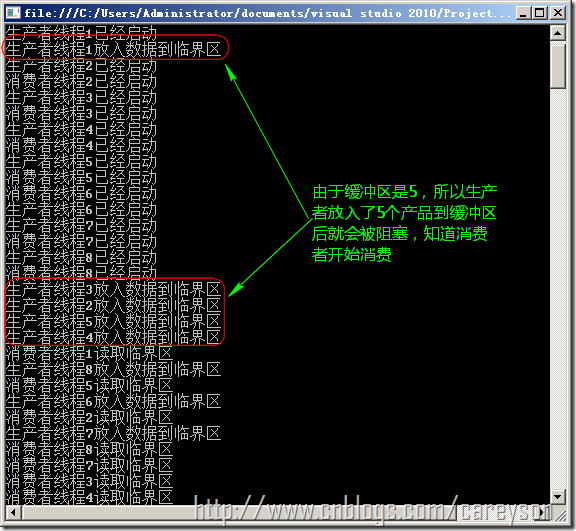
[](http://images.cnblogs.com/cnblogs_com/CareySon/201204/201204141750054782.png)

     图5.生产者消费者C#模拟结果

**读者--写者问题**

问题描述:

      一个数据文件或记录，统称数据对象，可被多个进程共享，其中有些进程只要求读称为"读者"，而另一些进程要求写或修改称为"写者"。

      规定:允许多个读者同时读一个共享对象，但禁止读者、写者同时访问一个共享对象，也禁止多个写者访问一个共享对象，否则将违反Bernstein并发执行条件。

    通过描述可以分析，这里的读者和写者是互斥的，而写者和写者也是互斥的，但读者之间并不互斥。

    由此我们可以设置3个变量，一个用来统计读者的数量，另外两个分别用于对读者数量读写的互斥，读者和读者写者和写者的互斥。如代码4所示。

class ReaderAndWriter

{

private static Mutex mut = new Mutex();//用于保护读者数量的互斥信号量

private static Mutex rw = new Mutex();//保证读者写者互斥的信号量

static int count = 0;//读者数量

static void Main()

{

Console.WriteLine("读者写者模拟......");

for (int i = 1; i < 6; i++)

{

Thread Thread1 = new Thread(new ThreadStart(Reader));

Thread1.Name = String.Format("读者线程{0}", i);

Thread1.Start();

}

Thread Thread2 = new Thread(new ThreadStart(writer));

Thread2.Name = String.Format("写者线程");

Thread2.Start();

Console.ReadKey();

}

private static void Reader()

{

mut.WaitOne();

if (count == 0)

{

rw.WaitOne();

}

count++;

mut.ReleaseMutex();

Thread.Sleep(new Random().Next(2000));//读取耗时1S

Console.WriteLine("读取完毕");

mut.WaitOne();

count--;

mut.ReleaseMutex();

if (count == 0)

{

rw.ReleaseMutex();

}

}

private static void writer()

{

rw.WaitOne();

Console.WriteLine("写入数据");

rw.ReleaseMutex();

}

   代码4.C#模拟读者和写者问题

    运行结果如图6所示。

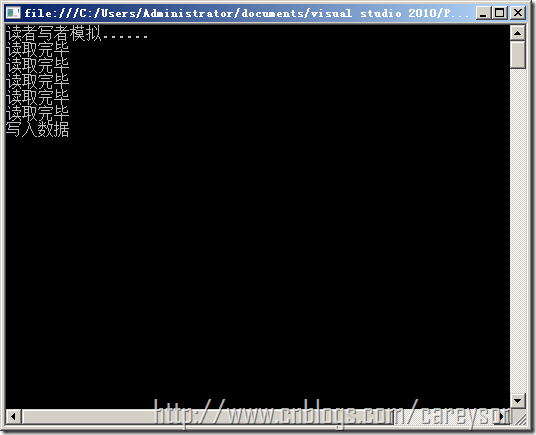
[](http://images.cnblogs.com/cnblogs_com/CareySon/201204/20120414175032653.png)

    图6.读者写者的运行结果

**哲学家进餐问题**

    问题描述:

    有五个哲学家，他们的生活方式是交替地进行思考和进餐。哲学家们公用一张圆桌，周围放有五把椅子，每人坐一把。在圆桌上有五个碗和五根筷子，当一个哲学家思考时，他不与其他人交谈，饥饿时便试图取用其左、右最靠近他的筷子，但他可能一根都拿不到。只有在他拿到两根筷子时，方能进餐，进餐完后，放下筷子又继续思考。

[](http://images.cnblogs.com/cnblogs_com/CareySon/201204/201204141750384868.png)

    图7.哲学家进餐问题

    根据问题描述,五个哲学家分别可以看作是五个进程。五只筷子分别看作是五个资源。只有当哲学家分别拥有左右的资源时，才得以进餐。如果不指定规则，当每个哲学家手中只拿了一只筷子时会造成死锁，从而五个哲学家都因为吃不到饭而饿死。因此我们的策略是让哲学家同时拿起两只筷子。因此我们需要对每个资源设置一个信号量，此外，还需要使得哲学家同时拿起两只筷子而设置一个互斥信号量，如代码5所示。

class philosopher

{

private static int[] chopstick=new int[5];//分别代表哲学家的5只筷子

private static Mutex eat = new Mutex();//用于保证哲学家同时拿起两双筷子

static void Main()

{

//初始设置所有筷子可用

for (int k = 1; k <= 5; k++)

{

chopstick[k - 1] = 1;

}

//每个哲学家轮流进餐一次

for(int i=1;i<=5;i++)

{

Thread Thread1 = new Thread(new ThreadStart(Philosophers));

Thread1.Name = i.ToString();

Thread1.Start();

}

Console.ReadKey();

}

private static void Philosophers()

{

//如果筷子不可用，则等待2秒

while (chopstick[int.Parse(Thread.CurrentThread.Name)-1] !=1 || chopstick[(int.Parse(Thread.CurrentThread.Name))%4]!=1)

{

Console.WriteLine("哲学家{0}正在等待", Thread.CurrentThread.Name);

Thread.Sleep(2000);

}

eat.WaitOne();

//同时拿起两双筷子

chopstick[int.Parse(Thread.CurrentThread.Name)-1] = 0;

chopstick[(int.Parse(Thread.CurrentThread.Name)) % 4] = 0;

eat.ReleaseMutex();

Thread.Sleep(1000);

Console.WriteLine("哲学家{0}正在用餐...",Thread.CurrentThread.Name);

//用餐完毕后放下筷子

chopstick[int.Parse(Thread.CurrentThread.Name)-1] = 1;

chopstick[(int.Parse(Thread.CurrentThread.Name)) % 4] = 1;

Console.WriteLine("哲学家{0}用餐完毕，继续思考", Thread.CurrentThread.Name);

}

}

    代码5.C#模拟哲学家用餐问题

    运行结果如图7所示。

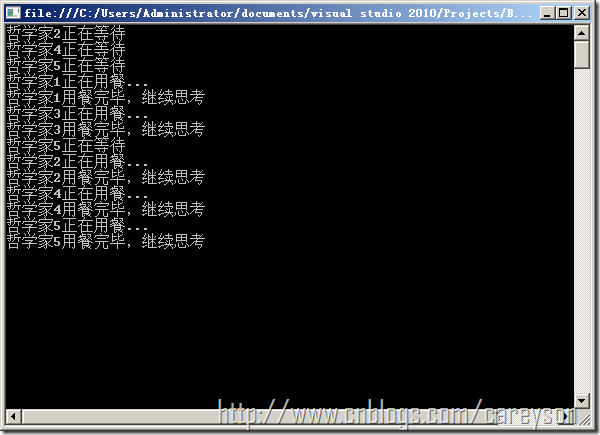
[](http://images.cnblogs.com/cnblogs_com/CareySon/201204/201204141750453120.png)

    图8.哲学家问题运行结果

### 总结

    本文介绍了进程的同步和互斥的概念，临界区的概念，以及实现进程同步互斥的方式，并解决了3种实现同步的经典问题，并给出了相应的C#模拟代码。操作系统对于进程的管理是是计算机编程的基础之一，因此掌握这个概念会使你的内功更上一层:-D

的关系。但他们对于临界区的访问又是互诉的关系。因此需要三个信号量empty和full用于同步缓冲区，而mut变量用于在访问缓冲区时是互诉的。

    利用C#模拟生产者和消费者的关系如代码3所示。

class ProducerAndCustomer

{

//临界区信号量

private static Mutex mut = new Mutex();

private static Semaphore empty = new Semaphore(5, 5);//空闲缓冲区

private static Semaphore full = new Semaphore(0, 5);

//生产者-消费者模拟

static void Main()

{

Console.WriteLine("生产者消费者模拟......");

for (int i = 1; i < 9; i++)

{

Thread Thread1 = new Thread(new ThreadStart(Producer));

Thread Thread2 = new Thread(new ThreadStart(Customer));

Thread1.Name = String.Format("生产者线程{0}", i);

Thread2.Name = String.Format("消费者线程{0}", i);

Thread1.Start();

Thread2.Start();

}

Console.ReadKey();

}

private static void Producer()

{

Console.WriteLine("{0}已经启动",Thread.CurrentThread.Name);

empty.WaitOne();//对empty进行P操作

mut.WaitOne();//对mut进行P操作

Console.WriteLine("{0}放入数据到临界区", Thread.CurrentThread.Name);

Thread.Sleep(1000);

mut.ReleaseMutex();//对mut进行V操作

full.Release();//对full进行V操作

}

private static void Customer()

{

Console.WriteLine("{0}已经启动", Thread.CurrentThread.Name);

Thread.Sleep(12000);

full.WaitOne();//对full进行P操作

mut.WaitOne();//对mut进行P操作

Console.WriteLine("{0}读取临界区", Thread.CurrentThread.Name);

mut.ReleaseMutex();//对mut进行V操作

empty.Release();//对empty进行V操作

}

}

   代码3.使用C#模拟生产者和消费者的关系

    运行结果如图5所示。

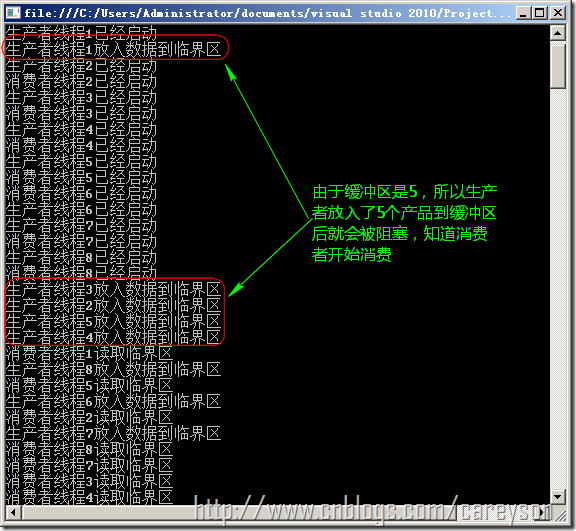
[](http://images.cnblogs.com/cnblogs_com/CareySon/201204/201204141750054782.png)

     图5.生产者消费者C#模拟结果

**读者--写者问题**

问题描述:

      一个数据文件或记录，统称数据对象，可被多个进程共享，其中有些进程只要求读称为"读者"，而另一些进程要求写或修改称为"写者"。

      规定:允许多个读者同时读一个共享对象，但禁止读者、写者同时访问一个共享对象，也禁止多个写者访问一个共享对象，否则将违反Bernstein并发执行条件。

    通过描述可以分析，这里的读者和写者是互诉的，而写者和写者也是互诉的，但读者之间并不互诉。

    由此我们可以设置3个变量，一个用来统计读者的数量，另外两个分别用于对读者数量读写的互诉，读者和读者写者和写者的互诉。如代码4所示。

class ReaderAndWriter

{

private static Mutex mut = new Mutex();//用于保护读者数量的互诉信号量

private static Mutex rw = new Mutex();//保证读者写者互诉的信号量

static int count = 0;//读者数量

static void Main()

{

Console.WriteLine("读者写者模拟......");

for (int i = 1; i < 6; i++)

{

Thread Thread1 = new Thread(new ThreadStart(Reader));

Thread1.Name = String.Format("读者线程{0}", i);

Thread1.Start();

}

Thread Thread2 = new Thread(new ThreadStart(writer));

Thread2.Name = String.Format("写者线程");

Thread2.Start();

Console.ReadKey();

}

private static void Reader()

{

mut.WaitOne();

if (count == 0)

{

rw.WaitOne();

}

count++;

mut.ReleaseMutex();

Thread.Sleep(new Random().Next(2000));//读取耗时1S

Console.WriteLine("读取完毕");

mut.WaitOne();

count--;

mut.ReleaseMutex();

if (count == 0)

{

rw.ReleaseMutex();

}

}

private static void writer()

{

rw.WaitOne();

Console.WriteLine("写入数据");

rw.ReleaseMutex();

}

   代码4.C#模拟读者和写者问题

    运行结果如图6所示。

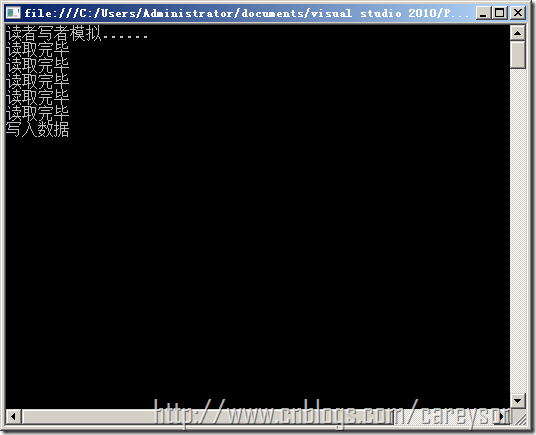
[](http://images.cnblogs.com/cnblogs_com/CareySon/201204/20120414175032653.png)

    图6.读者写者的运行结果

**哲学家进餐问题**

    问题描述:

    有五个哲学家，他们的生活方式是交替地进行思考和进餐。哲学家们公用一张圆桌，周围放有五把椅子，每人坐一把。在圆桌上有五个碗和五根筷子，当一个哲学家思考时，他不与其他人交谈，饥饿时便试图取用其左、右最靠近他的筷子，但他可能一根都拿不到。只有在他拿到两根筷子时，方能进餐，进餐完后，放下筷子又继续思考。

[](http://images.cnblogs.com/cnblogs_com/CareySon/201204/201204141750384868.png)

    图7.哲学家进餐问题

    根据问题描述,五个哲学家分别可以看作是五个进程。五只筷子分别看作是五个资源。只有当哲学家分别拥有左右的资源时，才得以进餐。如果不指定规则，当每个哲学家手中只拿了一只筷子时会造成死锁，从而五个哲学家都因为吃不到饭而饿死。因此我们的策略是让哲学家同时拿起两只筷子。因此我们需要对每个资源设置一个信号量，此外，还需要使得哲学家同时拿起两只筷子而设置一个互诉信号量，如代码5所示。

class philosopher

{

private static int[] chopstick=new int[5];//分别代表哲学家的5只筷子

private static Mutex eat = new Mutex();//用于保证哲学家同时拿起两双筷子

static void Main()

{

//初始设置所有筷子可用

for (int k = 1; k <= 5; k++)

{

chopstick[k - 1] = 1;

}

//每个哲学家轮流进餐一次

for(int i=1;i<=5;i++)

{

Thread Thread1 = new Thread(new ThreadStart(Philosophers));

Thread1.Name = i.ToString();

Thread1.Start();

}

Console.ReadKey();

}

private static void Philosophers()

{

//如果筷子不可用，则等待2秒

while (chopstick[int.Parse(Thread.CurrentThread.Name)-1] !=1 || chopstick[(int.Parse(Thread.CurrentThread.Name))%4]!=1)

{

Console.WriteLine("哲学家{0}正在等待", Thread.CurrentThread.Name);

Thread.Sleep(2000);

}

eat.WaitOne();

//同时拿起两双筷子

chopstick[int.Parse(Thread.CurrentThread.Name)-1] = 0;

chopstick[(int.Parse(Thread.CurrentThread.Name)) % 4] = 0;

eat.ReleaseMutex();

Thread.Sleep(1000);

Console.WriteLine("哲学家{0}正在用餐...",Thread.CurrentThread.Name);

//用餐完毕后放下筷子

chopstick[int.Parse(Thread.CurrentThread.Name)-1] = 1;

chopstick[(int.Parse(Thread.CurrentThread.Name)) % 4] = 1;

Console.WriteLine("哲学家{0}用餐完毕，继续思考", Thread.CurrentThread.Name);

}

}

    代码5.C#模拟哲学家用餐问题

    运行结果如图7所示。

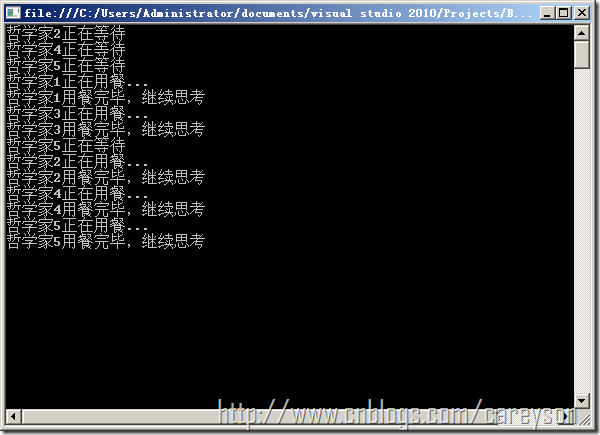
[](http://images.cnblogs.com/cnblogs_com/CareySon/201204/201204141750453120.png)

    图8.哲学家问题运行结果

### 总结

    本文介绍了进程的同步和互诉的概念，临界区的概念，以及实现进程同步互诉的方式，并解决了3种实现同步的经典问题，并给出了相应的C#模拟代码。操作系统对于进程的管理是是计算机编程的基础之一，因此掌握这个概念会使你的内功更上一层.

其中的“哲学家例子“，可以做为参考，在没有外在大系统规则的作用下，凭借“内部对象”的主管运动抢占资源，同时一个资源还是另外一个“对象”所需要，进行推演运算。主动推演运算。

C#的指针之美。

C#指针之美

<http://www.soaspx.com/dotnet/csharp/csharp_20120709_9393.html>