实 验 十　线程间通信与同步控制

10.1　实验目的

理解线程的并发模型，掌握多线程编写与启动方法。学习和掌握采用底层事件对象实现多线程同步控制的方法。

10.2　线程并发模型

Windows 平台程序运行是首先创建进程，进程仅是资源分配的单位，系统再创建线程对象，图10-1描述了 CPU 轮转产生的线程并发执行。

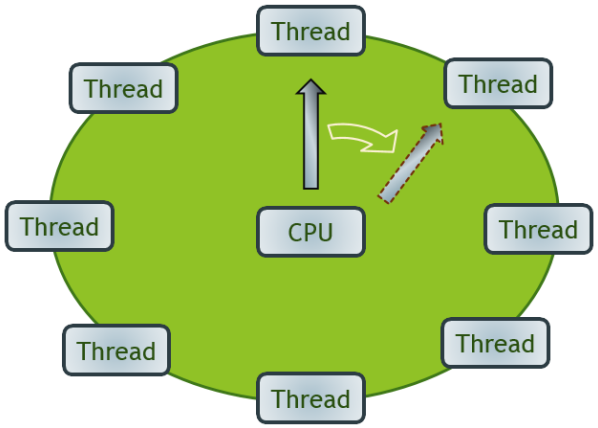


图 10-1 CPU 轮转与线程并发

多线程机制有较多优点，在具有一个处理器的计算机上，多个线程可以充分发挥机器计算能力，在一个线程计算时，还可接收用户的输入。线程机制使程序具有异步执行能力，程序还可以利用其他计算机的处理能力，与.NET Framework 远程处理或使用 ASP.NET 创建的 XML Web services 结合使用，如果程序需要进行大量的输入/输出工作，还可以使用 I/O 完成端口来提高应用程序的响应速度。多个线程适合的场合有下面情况：

1. 网络通信程序。
2. 与 Web 服务器和数据库操作。
3. 执行占用大量时间的操作。
4. 区分具有不同优先级的任务。例如，高优先级线程管理时间关键的任务，低优先级线程执行其他任务。
5. 界面线程快速响应用户输入，工作线程完成计算任务。

Windows 窗体进程包含一个主窗体线程和若干个工作线程，它们并发执行，线程间的协同工作需要同步通讯机制。程序设计者须将程序任务进行划分，主窗体线程仅限于获取输入与显示结果，工作线程执行消耗较多运算资源的任务。将耗时任务置于控件的事件代码中是糟糕的程序结构，窗体的消息循环因为无法继续不能及时响应后续消息，窗体会变得失去响应。

用户自定义消息处理方法是工作线程使用 SendMessage 接口向窗体发送消息，窗体线程利用消息循环机制来响应消息，这是一种非常有效的工作线程与窗体线程通信的方法。比较而言工作线程由用户设计循环机制作出响应，全部代码由用户设计，灵活性强设计难度大。

10.2.1 线程并发产生的问题

线程虽然能充分发挥机器性能，提供更好的用户交互性及实现任务的异步执行，但多线程带来如下的问题：

1. 线程如何实现稳妥结束。
2. 异步执行的任务数据同步控制。
3. 多线程间数据通信。
4. 多线程间资源争用。

系统为线程运行提供上下文信息使用内存，线程的并发需要有上下文切换过程，线程会带来资源特殊要求和潜在冲突。如果线程过多，系统管理线程的负担会加大，则其中大多数线程都不会产生明显的进度。如果大多数当前线程处于一个进程中，则其他进程中的线程的调度频率就会很低。多线程编程解决了吞吐量和响应性问题，但带来的新问题是死锁和争用条件。多个线程控制代码执行非常复杂，并可能产生许多 bug，当线程不能正常终结时，线程占用的内存等资源会影响系统的运行性能。

10.2.2 线程并发产生资源争用

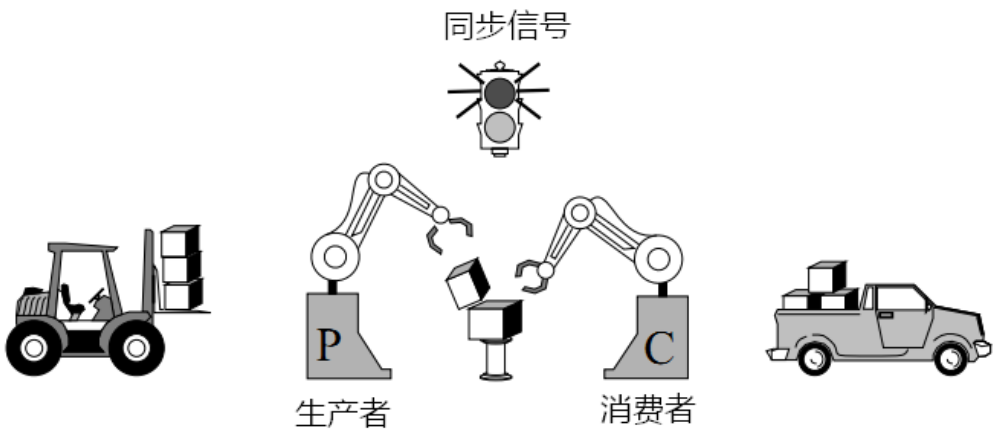


图 10-2 生产者和消费者

图10-2说明了典型的生产者消费者线程相互协作的工作模式，一个线程的输出是另一个线程的输入。中间结果变成两个线程的共享资源，它们对共享资源访问会引发冲突。为了避免冲突，访问共享资源必须进行同步控制。线程不恰当的异步推进顺序会导致一些问题，例如两个线程

不释放自己锁定的资源并试图访问对方已锁定的资源，造成任何两个线程都不能继续执行，这就是死锁。争用条件是指由于意外地出现对两个事件的执行时间的临界依赖性而发生反常的结果。程序在设计时应减少线程的数目，减少因同步资源造成的冲突。需要同步的资源包括：

1. 系统资源（如通信端口）。
2. 多个进程所共享的资源（如文件句柄）。
3. 由多个线程访问的单个应用程序域的资源（如全局、静态和实例字段）。

10.2.3 多线程控制同步对象类

.NET 平台中 System.Threading 命名空间包括了一些进行多线程编程的类和接口，表10-1提供了可用于协调多个线程之间的资源共享的同步对象。

表 10-1 同步控制类

类名	类说明
Mutex	提供对资源的独占访问，可用于同步不同进程中的线程。
Monitor	对特定对象获取锁和释放锁来公开同步访问代码区域的能力，它是局部的。
Interlocked	同步对多个线程共享的变量的访问的方法。
Semaphore	命名信号量（系统范围）或本地信号量。Windows 信号量是计数信号量，可用于控制对资源池的访问。
AutoResetEvent	本地等待处理事件，在释放了单个等待线程以后，该事件会在终止时自动重置，只能释放单个等待线程。
ManualResetEvent	表示一个本地等待处理事件，在已发事件信号后必须手动重置该事件，在对象保持已发信号状态期间，可以释放任意数目的等待线程。

终止多线程时要考虑以下准则：

1. 不使用 Thread.Abort 终止其他线程，也无法确定该线程已执行到哪个位置。
2. 不使用 Thread.Suspend 和 Thread.Resume 同步多个线程的活动。应该使用 Mutex、ManualResetEvent、AutoResetEvent 和 Monitor 类。
3. 主程序中控制通过同步对象，让辅助线程负责等待任务，执行任务，并在完成时通知程序的其他部分。

10.2.4 线程创建与终止

掌握使用同步对象是编写多线程程序的基础，线程的建立与启动需要下面几个步骤：

1. 编写线程代码。
2. 线程委托对象。
3. 设定线程优先级等属性。
4. 使用 Thread 对象启动线程。

在 C# 语言中启动线程可分为有参数和无参数两种，下面是无参方式启动的线程：

```
public delegate void ThreadStart();
```

传送给构造函数的参数必须是这种类型的委托。上面的例子中是 entryPoint，我们来看如何定义这个委托：

```
// 实际线程执行的方法必须定义为静态函数
Public static void thr_client_recv ()
{
// 接收线程入口
}
Thread clientThread = new Thread(clientWorkStart);
```

线程对象建立完成后，新线程实际上并没有执行任务，它只是在等待执行。我们需要显式地调用 Thread 对象的 Start() 方法来启动线程：

```
clientThread.Start();
```

此外还可以使用 Thread 对象的 Name 属性给线程赋予一个友好的名称。

图10-3表示了有参的线程启动方式。

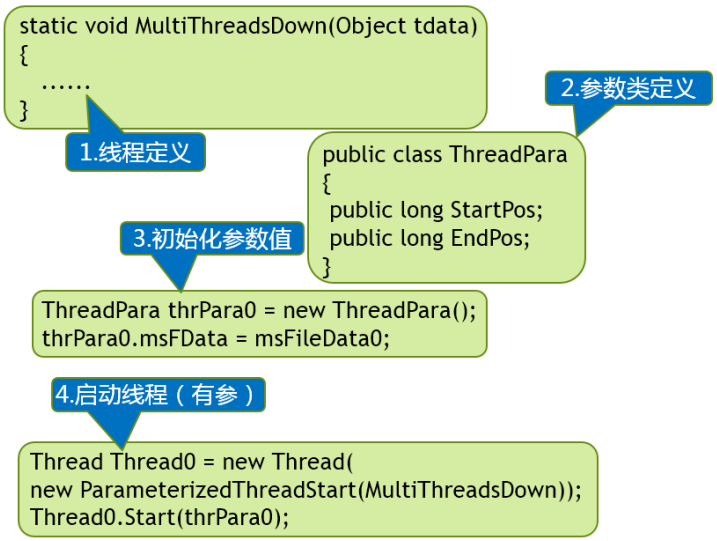


图 10-3 有参的线程启方式

10.3 多线程同步控制常见误区

多线程在最初设计时就要通过合理的分析，科学的构造，避免一些多线程设计过程中的常见误区。

10.3.1 使用 Kill 方法杀死线程

不能通过外部控制强行结束线程，使用 Kill 和 Abort 方法都不是正确终止线程的方法。使用 Kill 方法直接杀死线程如图10-4是不可取的，这会造成下面可能的不良后果，例如文件资源，数据库连接资源不能正常释放，网络连接的异常中断等。

1. 数据库连接资源没有正常释放。
2. 内存无法回收－内存泄漏。
3. 文件缓冲没写入－文件被破坏。
4. 文件句柄未回收－被占用。

图 10-4 杀死正在运行的线程

5. 共享资源的占用 (网络端口，管道，DLL)。
6. 网络连接的异常。

10.3.2 WaitOne 与 Sleep 方法比较

使用 Thread.Sleep 使线程等待的后果：等待期间线程不会有任何响应，也不能被杀死，极大影响执行性能，或造成死机现象。

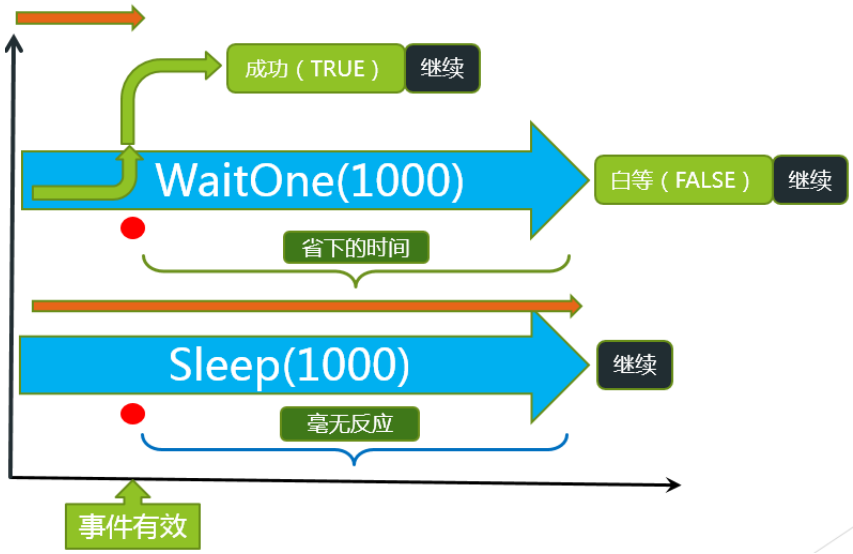


图 10-5 WaitOne 与 Sleep 方法比较

10.4 工作线程同步方法

线程结构的合理设计是线程稳健终结的前提，线程在能够灵敏感知外界对其的控制信息是非常重要的，这个特性与线程接收其它线程的同步信息本质是一样的。在设计线程代码时，构建具有事件检测的逻辑循环是关键。

互不相关的线程不存在同步控制的需求，当线程间发生生产者 -消费者关系时，同步控制能限制消费线程必须等待生产线程的数据。并发运行的线程由于推进速度不同，消费线程没有数据可用时进入" 无事可做" 的阶段。生产者的数据到达是不可预测的，消费线程设置循环机制反复查验数据的可用性，例如有程序设计者使用计时器执行定期查询，但是计时器在响应的及时性和资源利用率是矛盾的。较短的时间间隔会浪费较多的 CPU 运算能力，较长的时间间隔会使线程响应时间变长。

但是循环又极大浪费 CPU 运算能力，

多个相互联系的工作线程

本小节演示说明如何控制工作线程的方法，主要使用的是 ManualResetEvent 类对象，它同步控制效果良好，实现简单，而且系统开销较小。新建一个窗体应用程序，设位置为 D:\xue，名称为 thP；窗体界面设计参考图10-6所示。

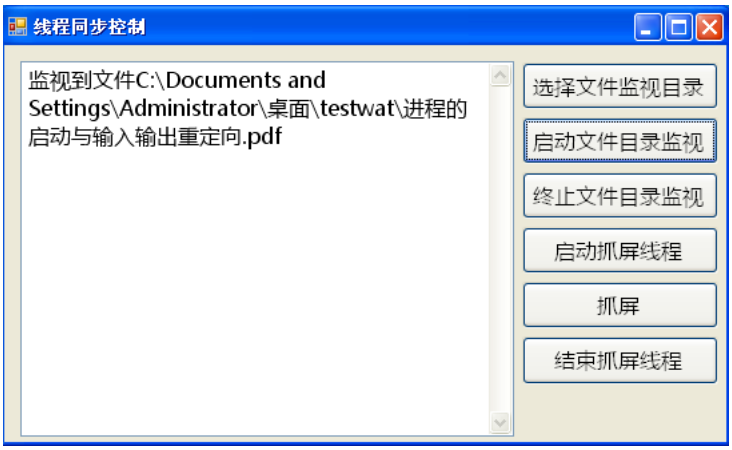


图 10-6 窗体与工作线程同步控制界面

10.4.1 文件目录监视

本小节完成对特定文件目录监视的任务，这个任务使用了工作线程，线程代码的编写与函数编写方式类似，使用 C# 代码启动线程需要通过线程委托的方式，启动线程的代码如下：

```
Thread workThread = new Thread(new ThreadStart(WatchDir));
workThread.IsBackground = true;
workThread.Start();
```

其中 WatchDir 即是线程函数名，这个线程要用到变量常量及内核函数声明如下：

```
//动态链接库引入
[DllImport("User32.dll", EntryPoint = "SendMessage")]
private static extern int SendMessage(
IntPtr hWnd, // handle to destination window
int Msg, // message
int wParam, // first message parameter
  int lParam // second message parameter
);
[DllImport("kernel32.dll")]
static extern int GetTickCount();
public const int WATCH_FILE = 0x500;
public static string w_dir;
public static ManualResetEvent e_wdirth_end;
public static IntPtr main_whandle;
```

```
public static string strfileinfo;
```

线程函数 WatchDir 的参考代码如下：

```
public static void WatchDir()
{
  long now_t = DateTime.Now.ToFileTime();
  DirectoryInfo d_info = new DirectoryInfo(w_dir);
  string new_filename;
  FileInfo[] f_ins = d_info.GetFiles();
  while (!e_wdirth_end.WaitOne(500))
  {
    for (int i = 0; i < f_ins.Length; i++)
    {
      now_t = DateTime.Now.ToFileTime();
      if (File.Exists(f_ins[i].FullName))
      {
        strfileinfo = string.Format(" 监视到文件 {0}\r\n",f_ins[i].FullName);
        SendMessage(main_whandle,WATCH_FILE,0,0);
        new_filename = w_dir + "\\ref\\" + now_t.ToString() + f_ins[i].Name;
        if (!File.Exists(new_filename))
        {
          File.Move(f_ins[i].FullName, new_filename);
        }
      }
    }
    f_ins = d_info.GetFiles();//重新获取新的目录信息
    }
}
```

在线程执行逻辑中，线程以循环的方式不停检测指定目录中的文件项，当有新文件出现时将文件重命名后移至 ref 子目录下。主窗体通过一个 ManualResetEvent 类型变量实现与工作线程的同步，e_wdirth_end 是这个 Event 对象名，e_wdirth_end.WaitOne(500) 这行代码在工作线程中的作用是使工作线程查看 Event 的状态，它的运行非常类似交通信号灯，如果 e_wdirth_end 事件状态没有被设为有效，这时可认为信号灯为红灯，线程将会在指定的 500 毫秒内中止；一旦事件状态被设为有效，这时可认为信号类变为绿灯，线程将被激活而执行后续代码。事件机制是 Windows 系统中重要的线程同步方式，线程的等待只消耗极少计算资源而获得较好的同步性能；休眠也能使线程等待，但是休眠的缺点是线程在休眠期间是无法被唤醒的；使用 ManualResetEvent 进行同步的方式要比系统忙等或者休眠的方式适应性强很多。

主窗体要能够接收线程发来的消息，需要进行消息处理函数的重载，参考代码如下：

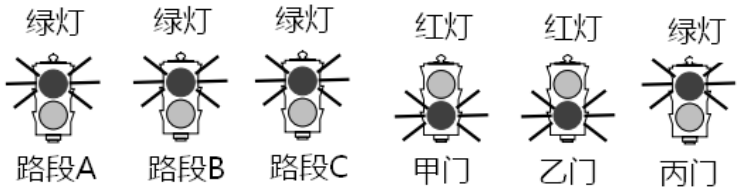
```
protected override void DefWndProc(ref Message m)
{
  switch(m.Msg)
```

```
        {
        case WATCH_FILE:
            textBox1.AppendText(strfileinfo);
            textBox1.ScrollToCaret();
            break;
        default:
            base.DefWndProc(ref m);
    break;
  }
  }
```

10.4.2 屏幕截屏线程

本小节使用工作线程实现对屏幕截屏，用户点击按钮时，线程将执行截屏任务并保存到图片文件中。工作线程函数名为 Capture_screen，线程参考代码如下：

```
  static void Capture_screen()
  {
    int s_wid = Screen.PrimaryScreen.Bounds.Width;
    int s_height = Screen.PrimaryScreen.Bounds.Height;
    Bitmap b_1 = new Bitmap(s_wid, s_height);
    Graphics g_ = Graphics.FromImage(b_1);
    String init_dir_fn = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
    String dest_fn = null;
    //用事件的方法来抓获图片
    //
    while (!capture_terminate_e.WaitOne(1, false))
    {
      if (capture_this_one_e.WaitOne(-1, false))
      {
        dest_fn = init_dir_fn;
        dest_fn += "\\bc\\";
        dest_fn += GetTickCount().ToString();
        dest_fn += "ab.bmp";
        g_.CopyFromScreen(0, 0, 0, 0, new Size(s_wid, s_height));
        b_1.Save(dest_fn, System.Drawing.Imaging.ImageFormat.Bmp);
        capture_this_one_e.Reset();
      }
    }
    g_.Dispose();
    b_1.Dispose();
```



(a) 事件同步对象的与关系　　(b) 事件同步对象的或关系

图 10-7 同步对象关系

}

在上面的示例代码中，capture_terminate_e 对象与 capture_this_one_e 对象都使用了 WaitOne 方法，方法的第一个参数值 -1 将使调用此方法的线程无限期等待指定事件状态到来。线程使用这两个对象时，它们是孤立的，即只能使用某一个同步对象，而在有些情况下需要考虑两个对象的联系，对多个事件同步对象有两种典型的联系，一种是或的关系，一种是与的关系。例如有一个路程由路段 A、路段 B 和路段 C 组成，必须三个路段的通行标志都为绿灯时整段路才可通行，同步对象间就是与关系，如图10.7a所示；而一个体育馆有三个门可进入，观众任选一个为绿灯的门即可进入，这三个门构成的同步对象对于观众就是或的关系，如图10.7b所示。

平台提供的 ManualResetEvent 类由 WaitHandle 类派生的孙子类，多个事件进行同步时应使用 WaitHandle 类的 WaitAll 方法或 WaitAny 方法，这两个方法在使用前需要将要同步的对象存入数组中，WaitAll 方法等待的事件状态是与的关系，这个方法的返回值是布尔值，表示所有的条件是否同时满足；WaitAny 方法等待的事件状态是或的关系，这个方法返回值类型是整型，如果有对象状态到来，返回值代表这个对象在等待的对象数组的索引值 (以 0 开始)，如果在指定时间段后没有事件状态到来，则返回值为 WaitHandle.WaitTimeout，这是个枚举量。原线程同时等待两个的事件，其一是主线程发来的拷屏可件命令，其二是主线程发来使其终止的事件，这两个事件有一定联系，把它们以或的关系进行同步将更适应实际运行需求。程序修改代码如下：

修改后的事件对象初始化代码：

```
public static ManualResetEvent capture_this_one_e;
public static ManualResetEvent capture_terminate_e;
public static ManualResetEvent[] me_cap = new ManualResetEvent[2];
private void button4_Click(object sender, EventArgs e)
{
  capture_terminate_e = new ManualResetEvent(false);
  capture_this_one_e = new ManualResetEvent(false);
  me_cap[0] = capture_terminate_e;
  me_cap[1] = capture_this_one_e;
  Thread workThread = new Thread(new ThreadStart(Capture_screen));
  workThread.IsBackground = true;
  workThread.Start();
```

```
}
```

修改后的线程代码：

```
static void Capture_screen()
{
  int s_wid = Screen.PrimaryScreen.Bounds.Width;
  int s_height = Screen.PrimaryScreen.Bounds.Height;
  Bitmap b_1 = new Bitmap(s_wid, s_height);
  Graphics g_ = Graphics.FromImage(b_1);
  String init_dir_fn = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
  String dest_fn = null;
  //用事件的方法来抓获图片
  int index = WaitHandle.WaitAny(me_cap, 500);
  while (index != 0)
  {
    if(index==1)
    {
      dest_fn = init_dir_fn;
      dest_fn += "\\bc\\";
      dest_fn += GetTickCount().ToString();
      dest_fn += "ab.bmp";
      g_.CopyFromScreen(0, 0, 0, 0, new Size(s_wid, s_height));
      b_1.Save(dest_fn, System.Drawing.Imaging.ImageFormat.Bmp);
      capture_this_one_e.Reset();
    }
    index = WaitHandle.WaitAny(me_cap, 500);
  }

  g_.Dispose();
  b_1.Dispose();
}
```

10.5 作业

1. 完善程序，实现 WaitAny 部分代码。