Threading in c#

Joseph albahri

به نام حضرت دوست که هر چه داریم از اوست

فصل اول

مرور مفاهیم

#C از اجرای موازی تکه کدهای یک برنامه با استفاده از سیستم Multithread حمایت می کند .یک Thread (نخ) یک مسیر اجرایی وابسته است که قابلیت اجرای همزمان با دیگر Thread ها را دارا می باشد .

هر برنامه ای که به زبان #C نوشته می شود با یک Thread (‘main’ Thread) شروع به کار می کند.این Thread به طور اتوماتیک توسط CLR(Common Language Runtime) و سیستم عامل ایجاد می شود.و با تولید Thread های دیگر یک ساختار multi-thread را دارا می باشیم.در اینجا ما یک مثال به همراه خروجی آنرا ملاحظه می کنیم:

توجه: تمام مثال ها با این فرض نوشته شده اند که فضای نام مربوطه وارد کد شده باشد

```
Using   System;
Using   System.Threading;
```

```
class ThreadTest
 {
   static void Main()
   {
       Thread  t = new Thread (WriteY);
       t.Start();                    // Run WriteY on the new thread
      while (true) Console.Write ("x");   // Write 'x' forever
   }
   static void WriteY()
   {
      while (true) Console.Write ("y");   // Write 'y' forever
   }
}
```

```
xxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  .....
```

Thread اصلی (Main) یک نخ جدید با نام t تولید می کند که این نخ تولید شده یک تابع که کاراکتر Y را در صفحه چاپ می کند را نشان می دهد.به طور همزمان نخ اصلی مکررا کاراکتر X را در صفحه چاپ می کند.

CLR به هر نخ مقدار حافظه مربوط به آن نخ را در حافظه پشته سیستم نسبت می دهد.این کار به دلیل جدا نگه داشتن متغیر های محلی انجام می گیرد.در مثال بعدی ما یک متد با یک متغیر محلی تعریف کرده ایم سپس متد مربوطه را به طور همزمان با نخ اصلی فراخوانی می کنیم.

```
static void Main()
{
  new Thread (Go).Start();      // Call Go() on a new thread
  Go();                         // Call Go() on the main thread
}

static void Go()
{
  // Declare and use a local variable - 'cycles'
  for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');
}
```

```
??????????
```

یک کپی جدا از متغیر Cycle در هر حافظه ی پشته ایجاد شده است و بنابراین خروجی به این صورت می باشد ، همانطور که می توان حدس زد تعداد ده علامت تعجب (?).

اگر نخ ها مرجعی به یک شی داشته باشند ، آنگاه اطلاعات مربوط به این شی را در اختیار یکدیگر می گذارند .مثال زیر را ملاحظه کنید:

```
class ThreadTest
{
  bool done;

  static void Main()
  {
    ThreadTest tt = new ThreadTest();   // Create a common instance
    new Thread (tt.Go).Start();
    tt.Go();
  }

  // Note that Go is now an instance method
  void Go()
  {
    if (!done)
   {
      done = true;
      Console.WriteLine ("Done");
    }
  }
}
```

چون هر دو نخ متد Go() را با استفاده از یک شی (ThreadTest) فراخوانی می کنند ، آنها عبارت Done را با یکدیگر به اشتراک گذاشته اند.این نتیجه به جای دو بار ، یک بار در صفحه چاپ خواهد شد.

```
Done
```

عناصر Static نوع دیگری از اشتراک منابع را بین Thread ها فراهم می کند.در زیر همان مثال بالا البته با یک عنصر Static نوشته شده است:

```
class ThreadTest
 {
   static bool done;    // Static fields are shared between all threads

   static void Main()
   {
      new Thread (Go).Start();
     Go();
   }

   static void Go()
   {
      if (!done)
      {
          done = true;
          Console.WriteLine ("Done");
      }
   }
}
```

هر دو این مثال ها مفهوم دیگری با نام *Thread safety* را نیز نشان می دهند.(حتی این مثال ها می توانند نشاندهنده ی فقدان این مفهوم هم باشند!).

خروجی در واقع نامعین خواهد بود: این امکان وجود دارد که عبارت Done دو بار در صفحه چاپ شود.در هر صورت اگر ما ترتیب عبارات را در متد Go() تغییر دهیم سپس تعداد فردی از عبارت Done دو بار در صفحه نمایش داده می شود

```
Done
Done    (usually)
```

مشکل اینجاست که یک نخ در حال اجرای دستور if را می باشد و نخ دیگر در حال اجرای دستور WriteLine می باشد- قبل از اینکه مقدار متغیر done فرصت true شدن را داشته باشد.

علاج کار استفاده از یک دستور قفل منحصر به فرد در هنگام خواندن و نوشتن در متغیر مربوطه می باشد.C# دستور lock را فقط برای این منظور فراهم کرده است :

```
class ThreadSafe
{
  static bool done;
  static object  locker = new object();

  static void Main()
  {
     new Thread (Go).Start();
     Go();
  }

  static void Go()
  {
    lock (locker)
    {
       if (!done)
       {
           Console.WriteLine ("Done");
           done = true;
       }
    }
```

```
    }
}
```

وقتی دو نخ به طور همزمان بر سر یک قفل (در این مثال locker) رقابت کنند ،تا زمانی که قفل آزاد نشود یک نخ یا منتظر می ماند یا مسدود می شود (block می شود) .در این موقعیت ما مطمئن می شویم که فقط یک نخ به بخش حیاتی کدمان دسترسی دارد و بنابراین عبارت Done فقط یک بار در صفحه چاپ می شود.کدی که به این طریق از نامعین بودن وضعیت حفظ می شود در اصطلاح *thread-safe* گفته می شود.

توقف موقت و یا مسدود شدن ، یک ویژگی ضروری در تطابق (*Synchronizing*) فعالیت نخ ها به شمار می رود.انتظار برای یک قفل منحصر به فرد یکی از دلایل مسدود شدن نخ هاست.دلیل دیگر برای این توقف زمانی پیش می آید که نخ بخواهد متوقف شود یا اصطلاحا برای مدتی عمل Sleep را انجام دهد :

```
Thread.Sleep (TimeSpan.FromSeconds (30));      // Block for 30 seconds
```

یک نخ می تواند با فراخوانی متد ()Join خود تا زمانی که دیگر نخ ها کارشان تمام شود منتظر آنها بماند.

```
Thread t = new Thread (Go);          // Assume Go is some static method
t.Start();
t.Join();                    // Wait (block) until thread t ends
```

یک نخ تا زمانی که مسدود شده است هیچ منبعی از منابع CPU را مصرف نمی کند.

Threading چه طور کار می کند

سیستم چند نخی به وسیله ی زمانبند نخ (thread scheduler) مدیریت می شود ، تابعی که CLR به عنوان مثال به سیستم عامل Delegate می کند.یک زمانبند نخ تضمین می کند که تمام نخ های فعال زمان اجرای متناسب خود را دارا می باشند، و اینکه نخ هایی که به عنوان مثال توسط یک قفل انحصاری یا ورودی کاربر به حالت انتظار درآمده اند یا مسدود شده اند ، زمان CPU را مصرف نمی کنند.

در یک کامپیوتر تک پردازنده ، یک زمانبند نخ عمل تقسیم زمان یا اصطلاحا time-slicing (اجرا را به سرعت بین هر نخ فعال سوییچ می کند) را برعهده دارد.این کار یک رفتار که دستخوش تغییر و تبدیل است را نتیجه می دهد ، مانند مثال های اولیه ما که هر بلاک تکرار کاراکتر های X و Y مطابق با تقسیم زمانی بود که برای آن نخ در نظر گرفته شده بود.در ویندوز XP یک تقسیم کننده ی زمان دارای سرعت انتخاب حوزه ای معادل با ده ها میلی ثانیه می باشد که این مقدار بسیار بزرگتر از میزان پردازش مورد نیاز CPU برای سوییچ کردن محتوا بین یک نخ و نخ دیگر می باشد (این زمان برای CPU در حوزه ی میکروثانیه قرار دارد).

در یک کاپیوتر چند پردازنده ، چند نخی (multithreading) به وسیله ی ترکیبی از (تقسیم زمانی) time-slicing و (همزمانی واقعی) genuine concurrency پیاده سازی می شود.در این مورد نخ های مختلف کد را در CPU های مختلف به طور همزمان اجرا می کنند.تقریبا مشخص است که هنوز هم مقداری تقسیم زمان به خاطر نیاز سیستم عامل به سرویس دهی به نخ های خودش وجود خواهد داشت.

اگر در اجرای یک نخ ، توسط یک فاکتور خارجی –هم چون تقسیم کننده ی زمان –وقفه ایجاد شود اصطلاحا گفته می شود که آن نخ از بقیه پیش دستی گرفته است (Preempt).در بیشتر موقعیت ها ، یک نخ هیچ کنترلی روی اینکه چه موقع و کجا از سایر نخ ها پیش می افتد ، ندارد.

Thread vs. Processes

همه ی نخ های درون یک برنامه منطقا درون یک Process قرار دارند.یک Process واحدی از سیستم عامل است که یک برنامه در آن اجرا می شود.

نخها شباهت هایی با Process ها دارند - برای نمونه ، Process ها با دیگر Process هایی که در کامپیوتر اجرا می شوند تقسیم بندی زمانی می شوند درست به همان شیوه ای که نخ ها در یک برنامه ی C# تقسیم بندی زمانی می شدند.تفاوت کلیدی که این دو با هم دارند در این مطلب است که Processها به طور کامل از یکدیگر جدا هستند؛ نخ ها حافظه ی heap را با دیگر نخ هایی که در همان برنامه در حال اجرا بودند به اشتراک می گذاشتند.این دلیلی است که نخ ها را مفید جلوه می دهد.یک نخ می تواند اطلاعات را در پس زمینه دریافت کند ، در حالی که دیگری در حال نمایش اطلاعات رسیده است.

زمان استفاده از نخ ها

یک برنامه عادی برای چند نخی وظیفه مصرف زمان (time-consuming) را در پس زمینه بر عهده دارد.نخ اصلی در حال اجرا باقی می ماند ، در حالی که نخ کارگر کار پس زمینه خود را انجام می دهد.در برنامه هایی که شامل فرم های ویندوز هستند یا برنامه های WPF ، اگر نخ اصلی مسئول انجام یک عملیات طولانی شود ، پیام های ماوس و صفحه کلید نمی تواند پردازش شود ، و برنامه غیر قابل پاسخ خواهد شد.به این دلیل ، بهتر آن است که اعمالی که زمان بیشتری را برای پردازش مصرف می کنند به عهده ی نخ های کارگر قرار دهیم ، حتی اگر نخ اصلی فقط مسئول نمایش پیغام "در حال پردازش.....لطفا صبر کنید" می باشد.با این کار ما تضمین می کنیم که برنامه هیچ وقت با پیغام Not Responding توسط سیستم عامل رو به رو نخواهد شد ، پیغام دروغینی که کاربر بالاجبار مجبور به توقف عمل پردازش می کند.خط مشی این نوع نمایش یک کلید برای انصراف از این پردازش را نیز فراهم می کند ، البته تا زمانی که عمل واقعی به درستی توسط نخ کارگر انجام شود.کلاس BackgroundWorker به استفاده از این الگو کمک می کند.

در برنامه هایی که واسط کاربری ندارند مانند سرویس های ویندوز ، چند نخی جلوه ی دیگری را هنگامی که عملی ذاتا زمان سیستم را مصرف می کند ، ایجاد می کند چون در این موقعیت این اعمال متنظر پاسخ از یک کامپیوتر دیگر می باشند.(مانند یک برنامه ی تحت سرور ، سرور پایگاه داده ، یا یک کلاینت).در اختیار داشتن یک نخ کارگر که وظیفه ای را انجام می دهد به معنی آن است که هر زمان آن نخ آزاد است تا عملی دیگر را انجام دهد.

دیگر استفاده ی چند نخی در متد هایی است که محاسبات متمرکز را انجام می دهند.چنین متد هایی در یک کامپیوتر چند پردازنده در صورتی که اعمال مربوطه میان چندین نخ تقسیم شوند ، با سرعت بیشتری می توانند اجرا شوند.(یکی از این نخ ها می تواند تعداد پردازنده های موجود را با استفاده از ویژگی *Environment.ProcessorCount* بدست آورد).

یک برنامه ی C# به دو طریق می تواند چند نخی شود.هم با ایجاد و اجرای مستقیم نخهای اضافی یا با استفاده از یکی از ویژگی های .Net Framework که به طور غیر مستقیم نخ ها را ایجاد می کند- مانند BackgroundWorker ، thread pooling ، یک threading timer ، یک سرور از راه دور یا یک وب سرویس یا یک برنامه ی ASP.NET.در مورد دومین شیوه هیچ انتخابی جز پذیرفتن چند نخی وجود ندارد.یک وب سرور تک نخی ASP.NET مطمئنا مناسب نخواهد بود- حتی اگر چنین چیزی ممکن باشد! خوشبختانه با برنامه های سروری Stateless ، چند نخی جدا ساده شده است. شاید یکی از موارد اهمیت چند نخی فراهم آودن مکانیزم های قفل حول اطلاعاتی که در متغیر های Static کش شده اند (Cach) ، باشد.

چه موقع نباید از نخ ها استفاده کرد

چند نخی همچنین اشکالات خود را نیز به همراه دارد.بزرگترین این اشکالات این مطلب است که چند نخی می تواند برنامه نویس را به سمت برنامه های بسیار پیچیده بکشاند.داشتن نخ های زیاد در برنامه به خودی خود پیچیدگی برنامه را زیاد نمی کند ؛ بلکه تقابل و بر هم کنش بین نخ ها این پیچیدگی را تولید می کند.این موضوع می تواند سوال زیر را تولید کند : اینکه آیا این تقابل ها عمدی انجام گرفته یا نه ؟ ، این موضوع می تواند چرخه توسعه طولانی ای را ایجاد کند ، همچنین در خطر بودن های مداوم و دوره ای برنامه و تولید باگ های بی پایان.به این دلیل ، ما باید بهایی را معادل با نگه داری چنین تقابل

ها و بر هم کنش هایی در زمان طراحی سیستم بپذیریم - یا اینکه اصلا از آنها استفاده نکنیم- مگر اینکه شما میل شدیدی به دوباره نوشتن و اجرا کردن برنامه داشته باشید!

اگر به طور افراطی از چند نخی استفاده شود مسایلی از قبیل بکار گرفتن حافظه برای منابع مورد استفاده نخ ها و نیز هزینه ی زمانی ای که CPU برای سوییچ کردن بین نخ ها استفاده می شود ، را ایجاد می کند.بخصوص زمانی که اعمال I/O سنگینی روی دیسک قرار است صورت پذیرد ، این مسئله می تواند با یک یا دو نخ کارگر که وظیفه انجام اعمال به صورت سلسله مراتبی را دارند ، به جای استفاده از گروه بسیاری از نخ ها که هر کدام یک کار را در همان زمان انجام می دهند، انجام گیرد.بعدا ما چگونگی تولید یک صف تولیدکننده/مشتری را توضیح خواهیم داد که فقط این کارایی را فراهم می کند.

ایجاد و شروع به کار کردن نخ ها

نخ ها با استفاده از سازنده ی کلاس Thread اجاد می شوند ، که به این سازنده یک نماینده (Delegate) از نوع ThreadStart فرستاده می شود – این نماینده نشاندهنده ی متدی است که باید اجرا شود.در پایین طریقه تعریف این نماینده نشان داده شده است :

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

فراخوانی متد Start نخ را اجرا می کند.نخ تا زمانی که متد مقداری را بازگرداند یا اینکه به پایان فراخوانی آن رسیده باشیم در حال اجرا باقی می ماند.در زیر مثالی آورده شده است که در آن استفاده از ThreadStart نیز گنجانده شده است.

```
class ThreadTest
{
   static void Main()
   {
      Thread t = new Thread (new ThreadStart (Go));
      t.Start();   // Run Go() on the new thread.
      Go();        // Simultaneously run Go() in the main thread.
   }
   static void Go()
   {
       Console.WriteLine ("hello!");
   }
}
```

در این مثال نخ t متد Go() را اجرا می کند در همان زمان نخ اصلی متد Go() را فراخوانی می کند و نتیجه نمایش دو عبارت hello می باشد.

```
hello!
hello !
```

یک نخ می تواند بسیار راحتتر نیز ساخته شود :

```
static void Main()
 {
    Thread t = new Thread (Go);    // No need to explicitly use ThreadStart
    t.Start();
    ...
}
static void Go() { ... }
```

در این موقعیت یک نماینده ی ThreadStart به طور اتوماتیک به وسیله کامپایلر به این شی اشاره داده می شود.نوع دیگر تولید یک نخ استفاده از متد های بی نام (anonymous method) است :

```
static void Main()
{
   Thread t = new Thread (delegate() { Console.WriteLine ("Hello!"); } );
   t.Start();
```

```
}
```

یک نخ دارای یک ویژگی (property) با نام IsAlive است که مقدار true را در خود ، بعد از فراخوانی متد Start() ، نگه می دارد و این مقدار تا زمانی که اجرای نخ پایان نیابد باقی می ماند.

اجرای یک نخ که پایان یافت ، دوباره نمی تواند شروع شود.

پاس دادن اطلاعات به ThreadStart

در مثال بالا ما می خواستیم تا خروجی ای که از هر نخ خارج می شد را بهتر ببینیم ، شاید با بزرگ نوشتن خروجی یکی از آنها.ما این کار را می توانیم با پاس دادن یک متغیر پرچم به متد Go() انجام دهیم : اما با این کار ما نمی توانیم از ThreadStart استفاده کنیم زیرا این نماینده هیچ آرگومان ورودی را نمی پذیرد.خوشبختانه ، Net Framework. نوع دیگری از نماینده با نام ParameterizedThreadStart را فراهم کرده است.که یک آرگومان از نوع object را قبول می کند :

```
Public delegate void ParameterizedThreadStart ( object obj );
```

در این صورت مثال قبلی شبیه زیر خواهد شد:

```
class ThreadTest
 {
   static void Main()
   {
      Thread t = new Thread (Go);
      t.Start (true);          // == Go (true)
      Go (false);
   }
   static void Go (object upperCase)
   {
      bool upper = (bool) upperCase;
      Console.WriteLine ( upper  ?  "HELLO!"  :  "hello!" );
   }
}
```

```
hello!
Hello!
```

در مثال بالا کامپایلر به طور اتوماتیک به یک نماینده از نوع ParameterizedThreadStart اشاره می کند ، زیرا متد Go() یک آرگومان از نوع object می پذیرد.ما همچنین می توانیم کد بالا را به این صورت بنویسیم :

```
Thread t = new Thread (new ParameterizedThreadStart (Go));
t.Start (true);
```

یک ویژگی استفاده از ParameterizedThreadStart این است که ما برای دستیابی به نتیجه ی درست باید آرگومانی که از نوع object تعریف کرده ایم را با استفاده از Type Casting تعیین نوع کنیم (در این موقعیت این نوع داده bool می باشد).

یک راه دیگر استفاده از متد های بی نام برای فراخوانی معمولی متد می باشد ، مانند زیر :

```
static void Main()
{
   Thread t = new Thread ( delegate() {  WriteText ("Hello");  } );
   t.Start();
}
static void WriteText (string text)
```

```
{
   Console.WriteLine (text);
}
```

فایده ی این کار در این است که متد هدف یعنی WriteText می تواند هر تعداد آرگومان که بخواهد قبول کند و هیچ عمل type Casting ای نیاز نخواهد بود.حتی می توان از بیرون نیز به این متد بی نام متغیر پاس داد همانطور که در زیر آمده است :

```
static void Main()
{
   string text = "Before";
   Thread t = new Thread (delegate() { WriteText (text); });
   text = "After";
   t.Start();
}
static void WriteText (string text)
 {
   Console.WriteLine (text);
}
```

```
After
```

اگر متد های بی نام با هر بار شروع یک نخ ویرایش شوند احتمال بی تناسبی را از تقابل های پیش بینی نشده به وسیله ی متغیرهای خارجی ایجاد می کنند. تقابل های نامزد (معمولاً به وسیله ی فیلدها) به طور کلی کافی به نظر می رسد! متغیر های خارجی بهترین رفتار را به عنوان یک متغیر read-only ، هنگامی که اجرای یک نخ شروع می شود ، دارند – مگر اینکه کسی بخواهد تا عمل قفل کردن یک نخ را در هر دو سو انجام دهد.

راه دیگر پاس دادن اطلاعات به یک نخ استفاده از متد static از یک شی می باشد . ویژگی های (property) شی ساخته شده می تواند به نخ بگوید که چه کاری می خواهد انجام دهد.همانطور که در زیر آمده است :

```
class ThreadTest
 {
    bool upper;

    static void Main()
    {
       ThreadTest   instance1 = new ThreadTest();
       instance1.upper  =  true;
       Thread t = new Thread (instance1.Go);
       t.Start();
       ThreadTest   instance2 = new ThreadTest();
       instance2.Go();        // Main thread – runs with upper=false
    }

    void Go()
    {
       Console.WriteLine ( upper  ?  "HELLO!"  :  "hello!" );
    }
}
```

نام گذاری نخ ها

یک نخ با استفاده از صفت Name خود می تواند نام گذاری شود.این یکی از بزرگترین ویژگی ها برای انجام عمل debug برنامه می باشد : همچنین ما قادر خواهیم بود تا با استفاده از Console.WriteLine نام نخ را مشاهده کنیم ، Microsoft Visual Studio نام نخ را برداشته و آنرا در

Debug Location Toolbar نمایش می دهد.نام یک نخ هر زمان که بخواهیم می تواند تنظیم شود – اما فقط برای یک بار – اگر بیشتر از یک بار بخواهیم این نام را تغییر دهیم با یک Exception روبه رو خواهیم شد.

نخ اصلی برنامه نیز می تواند یک نام به خود بگیرد در مثال زیر نخ اصلی به وسیله ی صفت ایستا(static) CurrentThread قابل دستیابی است :

```
class ThreadNaming
{
   static void Main()
   {
      Thread.CurrentThread.Name = "main";
      Thread worker = new Thread (Go);
      worker.Name = "worker";
      worker.Start();
      Go();
    }
   static void Go()
   {
      Console.WriteLine ("Hello from " + Thread.CurrentThread.Name);
    }
}
```

```
Hello from main
Hello from worker
```

نخ های پیش زمینه و پس زمینه

به طور پیش فرض ، نخ ها، نخ های پیش زمینه هستند ، به این معنا که ، تا زمانی که هر یک از این نخ ها در حال اجرا باشد برنامه را در حال اجرا نگه می دارند .C# همچنین از نخ های پس زمینه نیز پشتیبانی می کند ، که برنامه را تا زمانی که وجود داشته باشند در حال اجرا نگه نمی دارد – بعد از اینکه کار همه ی نخ های پیش زمینه تمام شد ، کار این نخ ها نیز پایان می پذیرد.

تغییر یک نخ از پیش زمینه به پس زمینه اولویت اجرای نخ را در زمانبند CPU تغییر نمی دهد.

صفت IsBackground یک نخ برای کنترل این وضعیت می باشد ، مانند مثال زیر :

```
class PriorityTest
 {
   static void Main (string[] args)
   {
      Thread worker = new Thread (delegate() { Console.ReadLine(); });
      if (args.Length > 0)
         worker.IsBackground = true;
      worker.Start();
   }
}
```

اگر برنامه با هیچ آرگومان ورودی ای فراخوانی شود ، نخ Worker در حالت پس زمینه اجرا شده و منتظر فشار داده شدن کلید Enter توسط کاربر می شود.در این هنگام نخ اصلی کار نخ اصلی تمام می شود اما برنامه هنوز در حال اجراست زیرا یک نخ پس زمینه در حال اجراست.

در غیر این صورت اگر آرگومانی به تابع Main() پاس داده شود ، Worker به حالت یک نخ پیش زمینه در می آید و برنامه به محض اینکه نخ اصلی پایان یابد ، تمام می شود.

وقتی یک نخ پیش زمینه به این طریق نابود می شود ، هر بلوک finally گیر خواهد افتاد و اجرا نخواهد شد ، و اجرا نشدن یک بلوک finally به طور کلی امری نامطلوب است. این ایده خوبی است که ما قبل از خروج از برنامه ، منتظر پایان کار هر نخ پس زمینه ای بمانیم – شاید با در نظر گرفتن یک زمان خروج(timeout) (این مطلب با فراخوانی متد Thread.Join امکان پذیر است).اگر به برخی دلایل یک نخ هرگز به پایان نرسید ، می توانید تا آنرا خاتمه دهید و اگر عمل خاتمه دادن شما به این برنامه با موفقیت انجام نگرفت، این نخ را ترک کنید و اجازه بدهید تا به وسیله process نابود شود.

پس زمینه کردن نخ کارگر می تواند فایده داشته باشد ،همیشه امکان فراخوانی آنها حتی وقتی که برنامه برای پایان آماده است وجود دارد.تفاوت را بررسی کنید – نخ های پیش زمینه با پایان برنامه ،به پایان خود نمی رسند – و از خروج برنامه جلوگیری می کنند. در یک برنامه ویندوزی یک نخ پیش زمینه که توسط process نابود شده است بسیار موذی است ، زیرا وقتی که کار نخ اصلی تمام می شود برنامه برای کاربر تمام شده نشان داده می شود اما پردازش این نخ همچنان باقی است. در Windows Task Manager ، این برنامه از بخش *Application* ناپدید می شود اگر چه نام فایل اجرایی همچنان در بخش Processes دیده می شود.اگر کاربر مستقیما این مکان را تعیین کرده و به این وسیله به برنامه خاتمه دهد ، وگرنه این نخ همچنان منابع مصرف کرده و حتی ممکن است از تولید نمونه ای جدید از برنامه جلوگیری کند یا در کارکرد آن تاثیر بگذارد.

یک دلیل رایج برای عدم موفقیت یک برنامه برای خروج درست ، وجود نخ های پیش زمینه ی فراموش شده است.

اولویت نخ

صفت Priority یک نخ تعیین کننده ی زمانی است که این نخ برای خود نسبت به دیگر نخ ها در همان Process می گیرد. و می تواند مقادیر زیر را دریافت کند :

```
enum ThreadPriority { Lowest , BelowNormal , Normal , AboveNormal , Highest }
```

این شیوه فقط برای نخ های چندگانه ای که به طور همزمان فعال هستند ، مناسب است .

تنظیم اولویت یک نخ به High به این معنی نیست که این نخ کار real-time را انجام می دهد. زیرا این نخ هنوز به وسیله ی اولویت پردازش برنامه محدود است. برای انجام عملی به صورت real-time ، کلاس Process در فضای نام System.Diagnostics به منظور بالا بردن اولویت یک process ، باید استفاده شود.همانطور که در زیر آمده است :

```
Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.High ;
```

ProcessPriorityClass.High در واقع نوعی از بالاترین اولویت پردازش است : Realtime.تنظیم اولویت یک process به Realtime ، به سیستم عامل می گوید که این process هرگز از دیگر نخ ها پیش دستی نخواهد کرد.اگر برنامه ی شما تصادفاً در یک حلقه بی نهایت قرار بگیرد انتظار این می رود که سیستم عامل نیز قفل کند.هیچ راهی جز فشار دادن کلید power شما را از این وضع نجات نخواهد داد ! به همین دلیل ، High به طور کلی به عنوان بالاترین اولویت پردازش در نظر گرفته می شود.

اگر برنامه ی Real-time یک واسط کاربری داشته باشد ، بالا بردن اولویت یک process امری نامطلوب است ، زیرا به روز رسانی صفحه نمایش زمان زیادی را از CPU می گیرد –سرعت یک کامپیوتر را پایین می آورد مخصوصاً اگر واسط گرافیکی بکار رفته دارای پیچیدگی باشد.پایین آوردن اولویت نخ اصلی –در مقایسه با بالا بردن اولویت process ها – تضمین می کند که نخ real-time به وسیله ی رسم دوباره ی صفحه از سایر نخ ها پیش دستی نمی کند ، اما از کند شدن سیستم جلوگیری نمی کند زیرا هنوز هم سیستم عامل به CPU پردازش های زیادی را تخصیص می دهد. راه حل ایده آل این است که عمل real-time و واسط گرافیکی در دو پردازش جدا (با اولویت های متفاوت) ، متصل به یکدیگر به وسیله Remoting یا حافظه اشتراکی ، قرار بگیرند .حافظه ی به اشتراک گذاشته شده نیازمند P/Invoking برای Win32 API(web-search CreateFileMapping and MapViewOfFile) می باشد.

11 مدیریت خطاها

وقتی یک نخ ایجاد می شود هر یک از ساختار های try/catch/finally ، زمانی که نخ شروع به اجرا کرد ، هیچ رابطه ای با یکدیگر ندارند .برنامه ی زیر را در نظر بگیرید:

```
public static void Main()
{
  try
  {
    new Thread (Go).Start();
  }
  catch (Exception ex)
  {
    // We'll never get here!
    Console.WriteLine ("Exception!");
  }

  static void Go()
  {
    throw null;
  }
}
```

عبارت try/catch در این مثال کاملاً بی فایده است و نخ جدید تولید شده با پیغام خطای مدیریت نشده ی NullReferencesExeption مواجه می شود . این موضوع وقتی جدی تر می شود که شما دارای نخی با مسیر اجرایی مستقل می باشید.علاج استفاده از این راه این است که متد وارده به نخ دارای مدیریت خطای خود باشد :

```
public static void Main()
{
   new Thread (Go).Start();
}

static void Go()
{
   try
   {
      ...
      throw null;      // this exception will get caught below
      ...
   }
   catch (Exception ex)
   {
       Typically log the exception, and/or signal another thread
   that we've come unstuck
       ...
   }
}
```

از .Net 2.0 به بعد یک خطای مدیریت نشده در هر نخی که کل برنامه را پایان می دهد ، به معنی نادیده گرفتن خطا به طور کامل است. از این رو یک بلوک try/catch در هر متدی که به نخ وارد می شود –حداقل در تولید برنامه –به منظور اجتناب از پایان های ناخواسته ی برنامه در موقعیت یک خطای مدیریت نشده ، ضروری است .این موضوع کمی مایه زحمت می باشد –مخصوصاً برای برنامه نویسان ویندوزی ، که معمولاً از مدیر خطای "global" برای انجام این موارد استفاده می کند.همان طور که در زیر نشان داده شده است :

```
using  System;
using  System.Threading;
using  System.Windows.Forms;
```

```
static class Program
 {
   static void Main()
   {
      Application.ThreadException += HandleError;
      Application.Run (new MainForm());
   }

   static void HandleError ( object  sender ,  ThreadExceptionEventArgs  e)
   {
       Log exception, then either exit the app or continue...
   }
}
```

رویداد Application.ThreadExeption زمانی که خطایی در برنامه اتفاق می افتد (یک Exception پرتاب می شود) روی می دهد و سرانجام به عنوان نتیجه ی یک پیام ویندوزی (برای مثال پیغام دریافتی از صفحه کلید یا ماوس یا Paint) فراخوانی می شود.

این روال به درستی کار می کند ، اما این کار یک اشتباه امنیتی را نشان می دهد –اینکه تمام خطاهای برنامه به وسیله مدیر خطای مرکزی گرفته می شود. خطاهایی که توسط نخ کارگر ایجاد می شود مثال خوبی از خطاهایی هستند که توسط Application.ThreadExeption گرفته نمی شوند (کد داخل تابع Main() نمونه ای دیگر از این مورد است –که شامل سازنده ی فرم اصلی است ، که قبل از شروع حلقه پیغام ویندوز اجرا می شود).

Net Framework. یک رویداد سطح پایین برای مدیریت خطای کلی فراهم کرده است : AppDomain.UnhandledException

این رویداد هنگامی که خطایی در هر نخی و در هر نوعی از برنامه (با واسط کاربری یا بدون آن) وجود داشته باشد ، رخ می دهد.در هر حال با اینکه این کار مکانیزم خوبی برای ثبت خطاهای مدیریت نشده ارائه می کند ، ولی هیچ راهی برای اجتناب از پایان برنامه فراهم نمی کند – ونیز هیچ راهی برای جلوگیری از نمایش dialog مربوط به مدیریت نادرست خطا توسط Net. نمی باشد.

در تولید برنامه ها ، مدیریت مستیم خطا ها در تمام متد های وارد شونده به یک نخ لازم است.ممکن است کسی این کار را با استفاده از یک کلاس کمکی ، مانند BackgroundWorker انجام دهد (این موضوع در بخش سه بحث می شود)

فصل دوم: پايه هاي هم زماني(Synchronize)

ضروریات هم زمانی

جدول زیر خلاصه ای از ابزارهای موجود در Net. برای هماهنگ کردن (یا synchronizing) فعالیت نخ ها می باشد:

متد های توقف سازی

ساختار	هدف
Sleep	توقف برای یک مدت معین

ساختار های قفل گذاری

ساختار	هدف	پردازش طولی؟	سرعت
Lock	تضمین می کند که فقط یک نخ می تواند به منابع سیستم یا بخشی از کد دسترسی پیدا کند.	خیر	سریع
Mutex	تضمین می کند که فقط یک نخ می تواند به منابع سیستم یا بخشی از کد دسترسی داشته باشد.برای جلوگیری از تولید نمونه های متعدد از ابتدا شروع برنامه استفاده می شود.	بله	متوسط
Semaphore	تضمین می کند که بیشتر از تعداد معینی از نخ ها نمی توانند به منبعی یا به تکه کدی دسترسی داشته باشند	بله	متوسط

ساختارهای علامت دهی

ساختار	هدف	پردازش طولی؟	سرعت
EventWaitHandle	تا زمانی که نخ حاضر یک علامت از دیگر نخ ها دریافت کند به آن اجازه انتظار می دهد	بلی	متوسط
Wait and pulse*	تا زمانی که با یک موقعیت مسدود کردن ملاقات نشده است به نخ اجازه انتظار می دهد.	خیر	متوسط

ساختار های هم زمانی *Non-blocking*

ساختار	هدف	پردازش طولی؟	سرعت
Interlocked*	انجام اعمال ساده و ریز non-blocking	بلی (حافظه اشتراک گذاشته شده را در نظر می گیرد)	بسیار سریع
Volatile*	به متغیر های خارج یک قفل اجازه دسترسی non-blocking من می دهد.	بلی (حافظه اشتراک گذاشته شده را در نظر می گیرد)	بسیار سریع

فیلد های ستاره دار در فصل چهار بحث می شوند.

انسداد (Blocking)

وقتی که یک نخ به عنوان نتیجه ی استفاده از یکی از ساختار های بالا به حال انتظار یا توقف می رود ، گفته می شود که این نخ قفل (Block) شده است . زمانی که نخ مسدود شد ،آن نخ به سرعت زمان اختصاص داده شده اش توسط CPU را رها می کند ، خاصیت WaitSleepJoin را به صفت ThreadState اضافه می کند و تا زمانی که از حالت انسداد بیرون نیاید دوباره زمان بندی نمی شود. بیرون آمدن از حالت انسداد در یکی از چهار حالت زیر اتفاق می افتد(کلید power کامپیوتر به حساب نمی آید !) :

- با پایان یافتن بلوک انسداد
- با پایان زمان timeout (اگر یک timeout تعیین شده بود)
- با تولید وقفه به وسیله ی Thread.Interrupt

- با خارج شدن از نخ به ویسله ی Thread.Abort

یک نخ در صورتی که اجرای آن به وسیله ی متد Suspend متوقف شده باشد ، مسدود شده فرض نمی شود.

Spinning و Sleeping

فراخوانی متد Thread.Sleep نخ جاری را برای یک مدت معین مسدود می کند (یا تا زمانی که Interrupt شود) :

```
static void Main()
{
  Thread.Sleep (0);                  // relinquish CPU time-slice
  Thread.Sleep (1000);               // sleep for 1000 milliseconds
  Thread.Sleep (TimeSpan.FromHours (1));  // sleep for 1 hour
  Thread.Sleep (Timeout.Infinite);       // sleep until interrupted
}
```

اگر بخواهیم با دقت بیشتری بررسی کنیم ، Thread.Sleep نخ را از CPU رها کرده ، وهیچ درخواستی برای آن نخ تا زمانی که مدت تعیین شده تمام نشود ، دوباره زمانبندی نمی شود.Thread.Sleep(0) باعث می شود که نخ جاری زمان گرفته شده ی CPU را فقط برای مدتی کمی رها کند تا در این مدت به دیگر نخ های فعال اجازه حاضر شدن در صف time-slicing به منظور اجرا را بدهد.

Thread.Sleep در میان دیگر متد های مسدود کننده ، یکتا است. چونکه پمپاژ پیام های ویندوز در یک برنامه ی ویندوزی را به حالت تعلیق در می آورد و یا در محیط های COM یک نخ ، برای هر مدل ساختمانی (Apartment model) تک نخ شده استفاده می شود.این موضوع به خاطر اثر کم با برنامه های ویندوزی است ، به همین خاطر هر عمل انسداد طولانی در نخ اصلی واسط کاربری ، برنامه را غیر قابل پاسخ خواهد کرد –از این رو کلاً از این شیوه اجتناب می شود – صرف نظر از اینکه آیا پمپاژ پیام ها از لحاظ فنی به حالت تعلیق در آمده باشد یا نه. این وضعیت در محیط های میزبانی COM کمی پیچیده تر می شود ،جایی که بعضی موقع عمل sleep مناسب است و پمپاژ پیام ها هم چنان وجود دارد.

کلاس Thread همچنین متدی با نام SpinWait فراهم کرده است ، که زمان گرفته شده توسط CPU را آزاد نمی کند و به جای آن در CPU می چرخد –معمولاً زمان CPU را برای انجام عملیات تکراری مشغول نگه می دارد.50 عمل تکرار ممکن است وقفه ای در حدود یک میکرو ثانیه ایجاد کند.اگر چه این رفتار به سرعت CPU و بارگذاری آن وابسته است. از لحاظ فنی متد SpinWait یک متد مسدود کننده نیست : یک نخ spin-waiting دارای ThreadState ای با نام WaitSleepJoin نمی باشد و به طور ناگهانی به وسیله ی دیگر نخ ها اصطلاحاً Interrupt می شود.SpinWait به ندرت استفاده می شود – هدف اصلی این متد منتظر ماندن برای یک منبع سیستمی است که انتظار می رود بسیار زود آماده شود (شاید در یک میکروثانیه) البته بدون فراخوانی متد sleep و هدر دادن زمان CPU با مجبور کردن یک نخ به منظور تغییر دادن آن.در هر حال این تکنیک فقط در کامپیتر های چند پردازنده ای دارای فایده است : در یک کامپیوتر تک پردازنده هیچ شانسی برای حالت یک منبع سیستمی برای عدم تغییر تا زمانی که time-sliceیک نخ spinning پایان یابد ، وجود ندارد –که هدف spinning را برای موقع شروع نقض می کند.

فراخوانی متد SpinWait اغلب یا برای مدت طولانی ای ، زمان CPU را هدر می دهد.

Blocking vs. Spinning

یک نخ با اجرای مستقیم spinning و با استفاده از یک حلقه ، می تواند منتظر یک وضعیت معین شود :

```
while ( !proceed ) ;
```

یا

```
while ( DateTime.Now < nextStartTime );
```

این کار زمان زیادی را از CPU هدر می دهد : تا آنجا که CLR و سیستم عامل تحت تاثیر قرار می گیرند ، نخ در حال انجام عملیات محاسباتی مهمی است بنابراین منابع سیستم را نیز در اختیار می گیرد! نخی که با این حالت حرکت می کند به عنوان یک مسدود شده به حساب نمی آید ، برخلاف نخی که برای یک EventWaitHandle منتظر می ماند (این ساختار معمولاً برای چنین وظایف علامت گذاری ای انتخاب می شود).

یک بی ثباتی در پیوند دو ترکیب Blocking و Spinning به صورت زیر می باشد :

```
while (!proceed) Thread.Sleep (x);    // "Spin-Sleeping!"
```

کاراکتر X بزرگتر، کارایی CPU بیشتر ؛ بعد از سبک سنگین کردن این نتایج میزان زمان بیکاری سیستم افزایش یافته است.هر چیزی در کد بالا موجب overhead ناچیزی در سیستم می شود – مگر اینکه وضعیت درون حلقه while واقعاً پیچیده باشد.

به جز زمان کم بیکاری سیستم ، ترکیب spinning و sleeping تقریباً خوب کار می کند (تابع مسئله ی همزمانی در پرچم Proceed که در فصل چهارم بحث خواهد شد می باشد). هنگامی که برنامه نویس بخواهد از یک ساختار پیچیده ی علامت گذاری استفاده کند ، شاید این مورد بزرگترین استفاده از این ساختار باشد!

Join (اتصال) کردن نخ ها به یکدیگر

شما می توانید عمل مسدود کردن را ، تا زمانی که اجرای نخی پایان یابد ، با استفاده از متد Join انجام دهید:

```
class JoinDemo
{
   static void Main()
   {
      Thread t = new Thread (delegate() { Console.ReadLine(); });
      t.Start();
      t.Join();    // Wait until thread t finishes
     Console.WriteLine ("Thread t's ReadLine complete!");
   }
}
```

متد Join همچنین یک آرگومان در واحد میلی ثانیه و نیز یک شی ای از کلاس TimeSpan را می پذیرد ، اگر اجرای نخ به پایان کار خود رسیده باشد متد Join مقدار false را باز می گرداند.Join با یک مقدار timeout مانند متد sleep کار می کند –در حقیقت هر دو تکه کد زیر کار یکسانی را انجام می دهد:

```
Thread.Sleep (1000);
Thread.CurrentThread.Join (1000);
```

تفاوت این دو کد در ساختارهای تک نخی برنامه های COM می باشد ، در واقع با تیزبینی و دقت در مفهوم عمل پمپاژ پیام ویندوز (windows message pumping) که قبلا توضیح داده شد ، می توان این تفاوت ها را مشاهده کرد : join عملیات پمپاژ پیام را تا زمانی که مسدود نشود در حال اجرا نگه می دارد ؛ sleep این پیام ها را به حالت تعلیق در می آورد.

قفل کردن (Locking) و امنیت نخ (Thread Safety)

قفل کردن بر دستیابی انحصاری تاکید دارد ، و برای تضمین اینکه فقط یک نخ می تواند وارد بخش های خاصی از یک کد در یک زمان شود ، استفاده می شود.برای مثال کلاس زیر را در نظر بگیرید :

```
class ThreadUnsafe
```

```
{
  static int val1, val2;

  static void Go()
  {
    if (val2 != 0)
      Console.WriteLine (val1 / val2);
    val2 = 0;
  }
}
```

این کد thread-safe نیست : اگر Go به وسیله دو نخ به طور همزمان فراخوانی می شد ، این امکان وجود داشت که تا یک خطای تقسیم بر صفر ایجاد شود –زیرا val2 با مقدار صفر در یک نخ پر می شود درست زمانی که نخ بعدی در حال اجرای دستور if و Console.WriteLine می باشد.

در این جا مثالی است که چه طور با استفاده از دستور lock می توان این مشکل را حل کرد :

```
class ThreadSafe
{
  static object locker = new object();
  static int val1, val2;

  static void Go()
  {
    lock (locker)
    {
      if (val2 != 0)
        Console.WriteLine (val1 / val2);
      val2 = 0;
    }
  }
}
```

فقط یک نخ می تواند شی همزمان کننده (synchronize) را قفل کند (در این موقعیت Locker) ، و تا زمانی که نخ جاری از حالت قفل در بیاید هر نخی مسدود می شود.اگر بیشتر از یک نخ در پشت دستور lock قرار گیرد آنها در صفی با نام Ready queue قرار می گیرد و به دستور lock اجازه می دهد که در یک الگوی First-come/First served ، به مدیریت نخ ها بپردازد.

بعضی موقع گفته می شود که قفل های انحصاری یک دستیابی ترتیبی از آنچه به وسیله ی قفل حفاظت شده را ایجاب می کنند ، زیرا با دستیابی به یک نخ تطابق و روی هم افتادن با دیگر نخ ها نمی تواند وجود داشته باشد.در این موقعیت ما از منطق بکار رفته در متد Go و همچنین متغیر های val1 و val2 استفاده می کنیم .

یک نخ مسدود شده تا زمانی که در یک قفل قرار دارد ، دارای مقدار WaitSleepJoin در متغیر ThreadState می باشد.بعدا ما در مورد اینکه چطور یک نخ که در این حالت مسدود شده است را بالاجبار به وسیله ی دیگر نخ ها با فراخوانی متد های Interrupt یا Abort ، آزاد کنیم.این کار جداً یک تکنیک سنگین است که از آن برای پایان یک نخ کارگر ممکن است ، استفاده شود.

دستور lock در C# در حقیقت یک میانبر برای فراخوانی متدهای Monitor.Enter و Monitor.Exit در یک بلوک try/finally ، می باشد. در زیر اتفاقی که واقعاً در متد Go می افتد نشان داده شده است :

```
Monitor.Enter (locker);
try
```

```
{
  if (val2 != 0)
     Console.WriteLine (val1 / val2);
  val2 = 0;
}
finally
{
   Monitor.Exit (locker);
}
```

فراخوانی متد Monitor.Exit بدون فراخوانی متد Monitor.Enter در همان شی تولید یک Exception می کند.

Monitor همچنین دارای یک متد TryEnter است که اجازه تعیین یک مدت زمان برای خروج (timeout) را می دهد –هم به صورت میلی ثانیه و هم به صورت TimeSpan .اگر یک قفل در ایجاد شود متد مقدار true را و اگر هیچ قفلی به خاطر متد timeout استفاده نشود مقدار false را باز می گرداند .TryEnter همچنین بدون آرگومان نیز می تواند فراخوانی شود ، که در این موقعیت قفل را امتحان می کند .اگر قفل نتواند فوراً مورد استفاده قرار بگیرد ، به سرعت timing out می شود.

انتخاب شی همزمان کننده

هر شی ای که برای نخ های موجود در برنامه قابل مشاهده باشد می تواند به عنوان شی همزمان کننده استفاده شود ، و این شی تابع یک وظیفه ی دشوار است : این شی می بایست از نوع مرجع (Reference Type) باشد.به منظور جلوگیری از تقابل های غیر عمدی که ممکن است به وسیله ی کد های خارجی این شی را قفل کند ، اکیداً تاکید می شود که شی هماهنگ کننده به صورت یک عنصر خصوصی (private) در داخل حوزه ی کلاس قرار گیرد.با توجه به این موضوع ، شی همزمان کننده می تواند با توجه به شی ای که از آن استفاده می کند ، دو تایی هم باشد به این صورت که در زیر نمایش داده شده است (دقت کنید که کلاس string یک نوع مرجع است):

```
class ThreadSafe
{
  List <string> list = new List <string>();

  void Test()
  {
    lock (list)
    {
       list.Add ("Item 1");
       ...
```

معمولاً یک فیلد اختصاصی (مانند Locker ، در مثال های قبلی) استفاده می شود ، زیرا با این کار می توان کنترل دقیقی روی حوزه و تکی بودن هر قفل ، داشت.در زیر استفاده از یک شی یا نوع داده را به عنوان یک شی همزمان کننده می بینید ، یعنی :

```
lock (this) { ... }

or:
lock (typeof (Widget))  { ... }    // For protecting access to statics
```

این موضوع کمی دلسرد کننده است زیرا با این کار ذاتاً یک حوزه عمومی (public) برای شی همزمان کننده فراهم کرده ایم.

عمل قفل گذاری دستیابی به شی همزمان کننده را به خودی خود محدود نمی کند. به عبارت دیگر ، x.ToString() مسدود نخواهد شد زیرا نخ دیگری دستور lock(x) را فراخوانی کرده است –به منظور اینکه عمل انسداد انجام گیرد هر دو نخ باید lock(x) را فراخوانی کنند .

قفل گذاری تو در تو

یک نخ مکرراً می تواند همان شی را قفل کند ، هم به وسیله ی فراخوانی Monitor.Enter و یا با به کار بردن تو در توی دستور lock .بعد از این شی با فراخوانی متد Monitor.Exit و یا با خروج از آخرین بلوک دستور lock می تواند از حالت انسداد بیرون بیاید. با این کار وقتی متدی ، متد دیگری را فراخوانی می کند ، مفهوم بهتری را از عمل انجام شده در اختیار ما قرار می دهد.

```
static object x = new object();

static void Main()
{
  lock (x)
  {
    Console.WriteLine ("I have the lock");
    Nest();
    Console.WriteLine ("I still have the lock");
  }
  در اینجا x آزاد می شود
}

static void Nest()
{
  lock (x)
  {
   ...
  }
  قفل آزاد می شود ؟ نه کاملاً!
}
```

یک نخ فقط در در اولین و یا در آخرین lock قفل می شود .

چه موقع شی ای را قفل کنیم

به عنوان یک قانون پایه ای ، هر متغیری که برای نخ های متعدد قابل دستیابی می باشد باید درون یک بلوک lock خوانده و نوشته شود .حتی در ساده ترین موقعیت –یک انتساب به یک متغیر –یک شی باید به عنوان همزمان کننده می بایست در نظر گرفته شود.در کلاس زیر نه متد Increment و نه متد Assign ، thread-safe نمی باشند.

```
class ThreadUnsafe {
  static int x;
  static void Increment() { x++; }
  static void Assign()    { x = 123; }
}
```

در زیر نمونه ی thread-safe کلاس بالا قرار داده شده است :

```
class ThreadUnsafe {
  static object locker = new object();
  static int x;

  static void Increment() { lock (locker) x++; }
  static void Assign()    { lock (locker) x = 123; }
}
```

میتوان برای قفل گذاری راه دیگری را نیز بیان کرد ، یک شی می تواند از ساختار همزمان کننده ی non-blocking در این موقعیت استفاده کند. این موضوع در فصل چهارم بحث می شود (به همراه آنها دلایل استفاده از چنین دستوراتی را برای همزمانی می بینید)

قفل گذاری و Atomicity

اگر گروهی از متغیر ها در یک بلوک lock خوانده و نوشته شوند ،گفته می شود که این متغیر ها اصطلاحاً به صورت *Atomic* خوانده و نوشته می شوند .متغیر های x و y را در مثال زیر در نظر بگیرید .این متغیرها در یک بلوک lock که بر روی شی locker اعمال شده است ، فقط خوانده می شوند و یا عمل انتساب در آنها صورت می گیرد.

```
lock (locker)
{
   if (x != 0)
      y /= x;
}
```

در این مثال گفته می شود که متغیر های x و y به صورت *Atomic* قابل دستیابی هستند ، زیرا بلوک کد نمی تواند به وسیله فعالیت دیگر نخ ها *تقسیم* شود یا اجرای آن نسبت به بقیه جلوتر بیافتد و روی این متغیرها تغییراتی را ایجاد کنند. با این کار شما هرگز پیغام خطای تقسیم بر صفر نمی گیرید ،به این دلیل که متغیر های x و y همیشه فقط درون همان بلوک *lock انحصاری* قابل دستیابی هستند.

ملاحضات کارایی

عمل قفل گذاری ، عمل سریعی است : یک قفل معمولا در ده ها نانوثانیه انجام می شود که چیزی حدود مقدار آن در صورت عدم استفاده از قفل است. اگر انسداد اتفاق بیفتد ، میزان سرعت بخش سوییچ کننده ی بین وظایف در کامپیوتر به سمت میکروثانیه حرکت می کند ، اگرچه این زمان شاید مربوط به میلی ثانیه های قبل از زمانبندی دوباره نخ ها باشد.این موضوع شاید در مقابل overhead های چند ساعته ، که ممکن است از قفل گذاری به هنگام نیاز آن جلوگیری کند! ، ناچیز باشد.

اگر به طور نادرست از قفل گذاری استفاده شود می تواند نتیجه ای ناسازگار با چیزی که ما می خواهیم را تولید کند –مثلا بی خاصیت کردن عمل همزمانی ، تولید بن بست و گردش قفل(race lock).

کم نیرو کردن عمل همزمانی زمانی اتفاق می افتد که تعداد خطوط کد زیادی در بلوک lock قرار داده شده باشد ، که این امر باعث می شود دیگر نخ ها مسدود شوند در حالی که اصلا نیازی به انسداد آنها نیست. یک بن بست زمانی است که هر یک از دو نخ به وسیله ی یک قفل توسط دیگری به حالت انتظار در آمده باشند ، بنابراین در این حالت هیچ یک از این دو نمی توانند کاری را انجام دهند. یک گردش قفل (race lock) زمانی اتفاق می افتد که برای هر دو نخ امکان استفاده از یک قفل وجود داشته باشد ، در این حالت اگر نخ مورد نظر این رقابت را نبرد برنامه متوقف خواهد شد. بن بست ها معمولاً بیشترین مشکل استفاده از تعداد بسیار اشیای همزمان کننده هستند. یک راه خوب برای حل این مشکل این است که قابلیت قفل گذاری را به تعداد کمتری از این اشیا بدهیم.

Thread Safety (امنیت نخ)

یک کد thread-safe ، کدی است که هیچ بی تکلیفی ای در سناریو بکار رفته دراستفاده از نخ های آن وجود نداشته باشد. امنیت نخ اساساً به وسیله قفل گذاری و کاهش تقابلات بین نخ ها فراهم می شود .

یک متد که از سناریو امنیت نخ پشتیبانی کند اصطلاحاً reentrant (روالی که می تواند توسط چندین برنامه مستقل بطور همزمان بکار برده شود) نامیده می شود. انواعی که برای اهداف کلی طراحی شده اند به ندرت در ساختار خود thread-safe هستند ، و این موضوع به دلایل زیر می باشد :

- هزینه ی توسعه ی نرم افزار در یک محیط کاملاً thread-safe می تواند دارای اهمیت باشد ، به ویژه اگر یک نوع داده فیلد های زیادی را به همراه داشته باشد (هر فیلد به طور ذاتی در یک محیط چند نخی دلخواه دارای تقابل می باشد)

- امنیت نخ می تواند شامل هزینه ی کارکرد شود (البته تا اندازه ای این هزینه قابل پرداخت است ،آیا این نوع داده واقعا برای کاربرد در نخ های متعدد استفاده می شود یا نه ؟)
- یک نوع داده ی thread-safe لزوماً نمی تواند برنامه را وادار به استفاده از قابلیت thread-safe خود کند –و بعضی موقع این کار می تواند تولید افزونگی اجرای دستور عمل در برنامه کند.

از این رو thread-safe فقط در جایی که به آن نیاز است پیاده سازی می شود ، مثلاً به منظور مدیریت یک سیستم چند نخی ویژه.

در هر حال راه های کمی برای فریب دادن ، ناشی از بکار بردن کلاس های پیچیده و بزرگ که امنیت را در یک محیط چند نخی تامین می کند ، وجود دارد. یک راه ، از بین بردن تکه تکه شدن های برنامه با تولید بخش های بزرگ کد است –هر کدام به یک شی درونی دستیابی داشته باشند –درون قفل های انحصاری است –در سطحی بالا بر دستیابی ترتیبی تاکید کنیم.این تاکتیک استفاده از یک شی thread-unsafe را درون کدی که thread-safe شده است سخت می کند – همچنین فراهم آوردن همان قفل انحصاری که برای حفاظت از دستیابی به صفت ها و متد ها و فیلد های یک شی thread-unsafe استفاده می شد ، نیز معتبر می باشد.

گذشته از انواع داده ی اصلی ،انواع داده ای کمی در .NET Frameword وجود دارند که برای فقط دستیابی همزمان به فیلد های read-only ، thread-safe هستند. این مسئولیت بر عهده ی تولید کننده ی نرم افزار است تا امنیت نخ را در برنامه وارد کند –مثلاً با استفاده از قفل های انحصاری.

حقه دیگری که می توان زد ، این است که تقابل نخ ها را با کاهش منابع اشتراکی نخ ها ، به حداقل برسانیم .این خط مشی یکی از بهترین ایده هاست و به طور غیر مستقیم در برنامه های Stateless و وب سرور ها استفاده می شود.زمانی که چند درخواست از طرف کلاینت ها همزمان فرا می رسد ، هر درخواست در نخ مربوط به خود قرار می گیرد (توسط ASP.NET ، سرویس های وب یا معماران از راه دور –Remoting architecture–) ،و این به این معنی است که متدی که این تکنولوژی ها استفاده می کنند thread-safe می باشد. یک طراحی stateless (به خاطر قابلیت مقیاس پذیری آن مورد توجه است) فی نفسه امکان تقابل را محدود می کند ، زیرا در غیر این صورت کلاس ها قادر نیستند تا اطلاعات بین هر درخواست را درست نگه داری کنند.تقابل نخ برای فیلدهای static نیز محدود می باشد ، ممکن است کسی آنرا تولید کند –شاید برای اهداف caching که معمولاً داده ها در حافظه انجام می دهند –و نیز در فراهم آوردن زیرساختهایی مانند تعیین هویت (authentication) و auditing در سرویس ها.

Thread-safety و انواع داده ها در .NET Framework

قفل گذاری می تواند کد thread-unsafe را به یک کد thread-safe تبدیل کند. یک مثال خوب با .Net Framework –تقریباً تمام نوع داده های غیر اصلی وقتی که از آنها شی ای ساخته می شود ،thread-safe نیستند ، و آنها می توانند در یک کد چند نخی استفاده شوند البته در صورتی که تمام دستیابی ها به وسیله lock حفاظت شده باشد. در زیر مثالی قرار دارد ، که دو نخ همزمان اطلاعاتی را به مجموعه ی List اضافه می کنند و سپس آنرا نمایش می دهند.

```
class ThreadSafe {
  static List <string> list = new List <string>();

  static void Main() {
    new Thread (AddItems).Start();
    new Thread (AddItems).Start();
  }

  static void AddItems() {
    for (int i = 0; i < 100; i++)
      lock (list)
        list.Add ("Item " + list.Count);

    string[] items;
```

```
    lock (list) items = list.ToArray();
    foreach (string s in items) Console.Write (s + " \t");
  }
}
```

نتیجه چیزی شبیه به شکل زیر است :

```
Item 0  Item 1  Item 2  Item 3  Item 4  Item 5  Item 6  Item 7  Item 8  Item 9
Item 0  Item 1  Item 2  Item 3  Item 4  Item 5  Item 6  Item 7  Item 8  Item 9
Item 10 Item 11 Item 12 Item 13 Item 14 Item 15 Item 16 Item 17 Item 18 Item 19
Item 20 Item 21 Item 22 Item 23 Item 24 Item 25 Item 26 Item 27 Item 28 Item 29
Item 30 Item 31 Item 32 Item 33 Item 34 Item 35 Item 36 Item 37 Item 38 Item 39
Item 40 Item 41 Item 42 Item 43 Item 44 Item 45 Item 46 Item 47 Item 48 Item 49
Item 50 Item 51 Item 52 Item 53 Item 54 Item 55 Item 56 Item 57 Item 58 Item 59
Item 60 Item 61 Item 62 Item 63 Item 64 Item 65 Item 66 Item 67 Item 68 Item 69
Item 70 Item 71 Item 72 Item 73 Item 74 Item 75 Item 76 Item 77 Item 78 Item 79
Item 80 Item 81 Item 82 Item 83 Item 84 Item 85 Item 86 Item 87 Item 88 Item 89
Item 90 Item 91 Item 92 Item 93 Item 94 Item 95 Item 96 Item 97 Item 98 Item 99
Item 100    Item 101    Item 102    Item 103    Item 104
Item 105    Item 106    Item 107    Item 108    Item 109
Item 110    Item 111    Item 112    Item 113    Item 114
Item 115    Item 116    Item 117    Item 118    Item 119
Item 120    Item 121    Item 122    Item 123    Item 124
Item 125    Item 126    Item 127    Item 128    Item 129
Item 130    Item 131    Item 132    Item 133    Item 134
Item 135    Item 136    Item 137    Item 138    Item 139
Item 140    Item 141    Item 142    Item 143    Item 144
Item 145    Item 146    Item 147    Item 148    Item 149
Item 150    Item 151    Item 152    Item 153    Item 154
Item 155    Item 156    Item 157    Item 158    Item 159
Item 160    Item 161    Item 162    Item 163    Item 164
Item 165    Item 166    Item 167    Item 168    Item 169
Item 170    Item 171    Item 172    Item 173    Item 174
Item 175    Item 176    Item 177    Item 178    Item 179
Item 180    Item 181    Item 182    Item 183    Item 184
Item 185    Item 186    Item 187    Item 188    Item 189
Item 190    Item 191    Item 192    Item 193    Item 194
Item 195    Item 196    Item 197    Item 198    Item 199
Item 10 Item 11 Item 12 Item 13 Item 14 Item 15 Item 16 Item 17 Item 18 Item 19
Item 20 Item 21 Item 22 Item 23 Item 24 Item 25 Item 26 Item 27 Item 28 Item 29
Item 30 Item 31 Item 32 Item 33 Item 34 Item 35 Item 36 Item 37 Item 38 Item 39
Item 40 Item 41 Item 42 Item 43 Item 44 Item 45 Item 46 Item 47 Item 48 Item 49
Item 50 Item 51 Item 52 Item 53 Item 54 Item 55 Item 56 Item 57 Item 58 Item 59
Item 60 Item 61 Item 62 Item 63 Item 64 Item 65 Item 66 Item 67 Item 68 Item 69
Item 70 Item 71 Item 72 Item 73 Item 74 Item 75 Item 76 Item 77 Item 78 Item 79
Item 80 Item 81 Item 82 Item 83 Item 84 Item 85 Item 86 Item 87 Item 88 Item 89
Item 90 Item 91 Item 92 Item 93 Item 94 Item 95 Item 96 Item 97 Item 98 Item 99
```

در این مثال ما متغیر list را قفل کرده ایم ، که برای این مثال گزینه ی مناسبی بود. اگر ما دو لیست مرتبط به یکدیگر داشتیم ، ما نیاز داریم تا شی دیگری را قفل کنیم –اگر هیچ یک از دو لیست نامزد قفل شدن نشد ،شاید یک متغیر جدا از دو لیست را قفل می کردیم.

عمل نمایش اطلاعات در مجموعه های .Net نیز یک عمل thread-unsfase است ، زیرا نخ دیگری در حین نمایش اطلاعات می تواند داده ای را به مجموعه اضافه کند ، و این کار تولید یک Exception می کند. در این مثال ما ابتدا اطلاعات را در آرایه ای کپی کردیم و قفل گذاری را برای زمان نمایش اطلاعات انجام دادیم. اگر کاری که ما در حین نمایش اطلاعات انجام می دهیم ذاتاً زمانی را مصرف می کند ، با این کار از نگه داشتن افراطی قفل جلوگیری می کند.

در ادامه یک فرض جالب را می بینیم :فرض کنید که کلاس List واقعا thread-safe بود. این کار چه چیزی را حل خواهد کرد؟

در واقع مشکلات بسیار کمی را می توانست حل کند! برای نشان دادن این موضوع ، ما می خواهیم اطلاعاتی را به این لیست امن فرضی خود اضافه کنیم ، همانطور که در زیر نشان داده شده است :

```
if (!myList.Contains (newItem)) myList.Add (newItem);
```

آیا لیست thread-safe است یا خیر ، در این دستور واقعاً این طور نیست ! به منظور جلوگیری از پیشدستی برنامه برای انجام عمل مقایسه و اضافه کردن اطلاعات به لیست ، تمام دستور if می بایست در یک بلوک lock قرار داده شود. این قفل هر جایی که ما نیاز به اصلاح اطلاعات درون لیست داشتیم ، نیز باید قرار داده شود. برای مثال کد زیر نیز باید در یک بلوک lock قرار داده شود –در یک قفل یکسان :

```
myList.Clear();
```

با این کار ما تضمین می کنیم که این دستور از دستور قبلی خود پیش دستی نمی کند. به عبارت دیگر ، ما می بایست دقیقاً به همان ترتیب که برای کلاس thread-unsafe خود رفتار کردیم ، دستورات را قفل کنیم . در نتیجه قرار دادن امنیت نخ در این کلاس ها فقط می تواند وقت برنامه نویس را تلف کند!

می توان این موضوع را وقتی که می خواهیم کامپوننت های خودمان را بنویسیم مساله ی قابل بحثی باشد –چرا ما باید یک کد امن را بنویسیم در حالی که این کار باعث افزونگی کد در برنامه خواهد شد.

نظری متقابل با بالا نیز وجود دارد : قرار دادن اشیا در یک بلوک lock فقط زمانی کار می کند که همه ی نخ هایی که همزمان اجرا می شوند ، از آن آگاه باشند و از آن استفاده کنند –که اگر کد در حوزه بزرگی نوشته شده باشد در این شرایط قرار نمی گیرد.بدترین اتفاق که به صورت غیر عمدی پیش می آید قرار دادن اعضای static در یک نوع عمومی می باشد. برای مثال ، صفت static ، DateTime.Now را در ساختمان DateTime در نظر بگیرید ، این صفت thread-safe نیست ، و فراخوانی همزمان دو نمونه از این ها می تواند خروجی اشتباه یا یک Exception تولید کند. تنها راه چاره برای حل این مشکل با یک قفل خارجی ، قفل کردن خود نوع داده در برنامه می باشد – lock(typeof(DateTime)) - که فقط زمانی کار خواهد کرد که همه ی برنامه نویسان با این کار موافق باشند. و این موافقت واقعاً بعید است ، زیرا قفل کردن یک نوع داده توسط بسیاری از برنامه نویسان ، چیز بدی در نظر گرفته می شود!

به این دلیل تضمین می شود که اعضای static در ساختمان DateTime ، thread-safe هستند. این موضوع که –اعضای استاتیک همگی thread-safe هستند در حالی که اعضایی که در سطح شی استفاده می شوند (اعضای عمومی) این گونه نیستند –یک الگوی کلی در .NET Framework است.پیروی از این موضوع نوشتن انواع تولیدی خودمان را نیز می تواند تحت تاثیر قرار دهد.

وقتی در حال نوشتن کامپوننت هایی برای مصرف خودمان هستیم ، خط مشی مناسب این است که نباید مانع اعمال امنیت نخ در آنها شد. یعنی مراقب استفاده از اعضای استاتیک در برنامه باشیم.

Abort و Interrupt

یک نخ مسدود شده به دو طریق می تواند ناگهان آزاد شود :

- به وسیله ی Thread.Interrupt
- به وسیله ی Thread.Abort

این کار باید توسط فعالیت دیگر نخ ها انجام گیرد ؛ زیرا نخی که به حالت انتظار در آمده است هیچ قدرتی برای انجام این کار ندارد.

Interrupt

فراخوانی متد Interrupt در یک نخ مسدود شده بالاجبار آنرا آزاد می کند ، و یک ThreadInterruptedException نیز پرتاب می کند ، همانطور که در زیر نشان داده شده است :

```
class Program
 {
   static void Main()
   {
     Thread t = new Thread (delegate() {
                                 try
                                 {
                                   Thread.Sleep (Timeout.Infinite);
                                 }
                                 catch (ThreadInterruptedException)
                                 {
                                   Console.Write ("Forcibly ");
                                 }
                                 Console.WriteLine ("Woken!");
                               }
                     );
   t.Start();
   t.Interrupt();
  }
}
```

```
Forcibly Worken!
```

تولید وقفه برای یک نخ آنرا فقط از انتظار حال حاضر (یا بعدی) آزاد می کند .این کار باعث پایان کار نخ نمی شود (البته اگر ThreadInterruptedExeption به درستی مدیریت نشود ، می توان این کار را انجام داد!)

اگر Interrupt برای نخی که مسدود نشده است فراخوانی شود ، نخ به اجرای خود تا رسیدن به سد بعدی ادامه می دهد ، در این موقعیت یک ThreadInterruptedException پرتاب می شود. این کار نیاز به امتحان کد زیر را برای ما رفع می کند :

```
if ( ( worker.ThreadState & ThreadState.WaitSleepJoin) > 0)
  worker.Interrupt() ;
```

کد بالا یک کد thread-safe نمی باشد ، زیرا امکان پیش دستی بین دستورات if و worker.Interrupt وجود دارد.

توقف دلخواهانه ی یک نخ خطرناک است ، زیرا ، هر محیط کاری یا هر متد third-party ای در فراخوانی پشته برنامه می تواند یک وقفه غیر منتظره دریافت کند. همه چیزی که این خواهد گرفت ، مربوط به نخ هایی است که مختصراً در یک قفل ساده یا در یک منبع همزمان کننده مسدود شده اند و در این حالت هیچ وقفه معوقی اتفاق نمی افتد. اگر متدی طوری طراحی شده باشد که وقفه تولید نکند ، اشیا بی استفاده می توانند ترک شوند و یا ممکن است منابع سیستم به طور کامل آزاد نشوند.

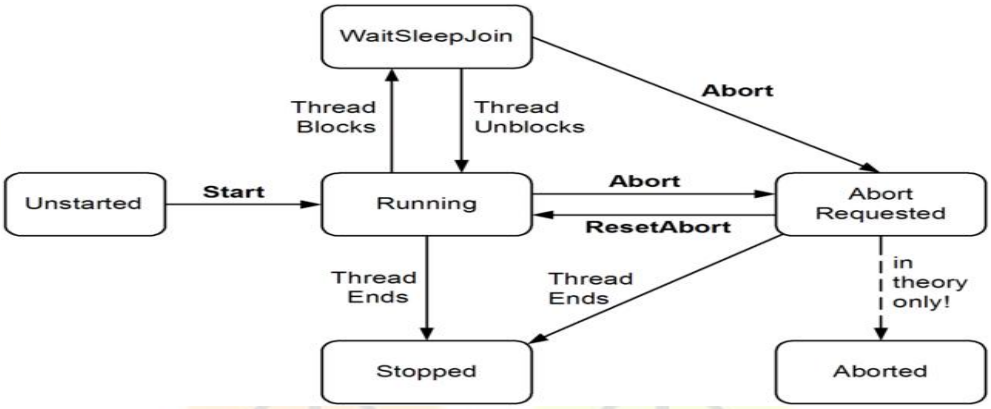
توقف یک نخ زمانی امن است که شما بدانید نخ دقیقاً در کجا قرار دارد.بعداً ما ساختارهای علامت دهی را که چنین ابزاری را برای ما فراهم می کنند را توضیح خواهیم داد.

Abort

یک نخ بالاجبار می تواند با متد Abort نیز آزاد شود. این متد اثری شبیه به فراخوانی متد Interrupt دارد ، به استثنای اینکه یک خطای ThreadAbortException به جای ThreadInterruptedException تولید می شود. بنابراین خطا دوباره در پایان بلوک catch می تواند پرتاب شود ، مگر اینکه درون بلوک catch یک فراخوانی برای متد Thread.ResetAbort صورت پذیرد. در این موقعیت نخ دارای حالت AbortRequested در صفت ThreadState خود می باشد.

به هر حال تفاوت اساسی بین متد Interrupt و Abort در اتفاقی است که هنگام فراخوانی یک نخ که مسدود نیست ، وجود دارد. متد Interrupt منتظرمی ماند تا نخ به سدهای بعدی برسد ، در حای که متد Abort یک استثنا در جایی که نخ در حال اجراست پرتاب می کند. پایان دادن(فراخوانی متد Abort) به یک نخ مسدود نشده می تواند اثرات مهمی داشته باشد. جزییات این موارد در بخش بعدی (Aborting Thread) بحث خواهد شد.

حالت نخ



شکل 1- نمودار حالت نخ

ما می توانیم به وسیله ی صفت ThreadState در حالت یک نخ جستجو کنیم. شکل 1 یک لایه از نوع داده شمارشی (Enumerate) ThreadState را به ما نشان می دهد.در این قسمت ThreadState بد طراحی شده است ، بدین ترتیب که این نمودار شامل ترکیب سه لایه حالت که از پرچم های bitwise استفاده می کنند ، می باشد ، اعضای هر لایه متقابلاً منحصر به فرد هستند. در زیر این سه لایه توضیح داده شده است :

- حالت اجرا (running) / انسداد (blocking) / پایان (aborting)
- حالات پس زمینه / پیش زمینه (ThreadState.Backgroud)
- حرکت به سمت تعلیق به وسیله ی متد قدیمی و از رده خارج Suspend (ThreadState.SuspendRequest و ThreadState.Suspended)

در کل ، ThreadState ترکیب بیتی یکی یا هیچ یک از اعضای هر لایه است! در زیر مثالهای ساده ای از ThreadState وجود دارد :

```
Unstarted
Running
WaitSleepJoin
Background, Unstarted
SuspendRequested, Background, WaitSleepJoin
```

این نوع داده (ThreadState) دو عضو دارد که هرگز استفاده نمی شوند ، حداقل در پیاده سازی حال حاضر CLR :

StopRequest و Aborted.

برای پیچیده تر کردن موضوع ، ThreadState.Running دارای مقدار صفر در این مجموعه می باشد ، بنابراین آزمون زیر به درستی کار نخواهد کرد :

```
if ( ( t.ThreadState & ThreadState.Running ) > 0) ...
```

بجای آزمایش اجرای یک نخ با اعمال محدودیت یا چک کردن نوبتی آن ، می توان از صفت IsAlive استفاده کرد. به هر حال IsAlive نیز ممکن است آن چیزی شما بخواهید نباشد. این صفت اگر نخ مسدود شده باشد یا به حالت تعلیق در آمده باشد مقدار true را بازمی گرداند (تنها زمانی که این صفت مقدار false را بازمیگرداند قبل از اجرای نخ و یا بعد از پایان کار نخ است).

فرض کنید کسی به سلامت از متد های قدیمی Suspend و Resume رد شود ، این فرد می تواند یک متد کمکی بنویسد که تمام اعضا به جز اعضای موجود در لایه اول را حذف کند ، و اجازه ی آزمایش تساوی را به راحتی فراهم می کند. در حالت استفاده از نخ پس زمینه می توان مستقلاً از صفت IsBackground استفاده کرد ، بنابراین اولین لایه واقعاً اطلاعات درستی را در بر دارد :

```
public static ThreadState SimpleThreadState (ThreadState ts)
{
 return ts & (ThreadState.Aborted | ThreadState.AbortRequested |
          ThreadState.Stopped | ThreadState.Unstarted |
          ThreadState.WaitSleepJoin);
}
```

ThreadState برای مقاصد debug و profiling ، بسیار گران بها است. با اینکه ThreadState بسیار کم مفید است ، در هر حال ، برای هماهنگ کردن نخ های متعدد مفید است ، زیرا هیچ مکانیزمی برای امتحان یک حالت ThreadState و سپس عمل بر روی آن اطلاعات بدون تغییر موقتی اطلاعات ThreadState ، وجود ندارد.

مدیریت انتظار

دستور lock (Monitor.Enter / Monitor.Exit) مثالی از یک ساختار همزمان کننده نخ است. مادامی که lock برای وادار کردن دستیابی انحصاری به یک منبع خاص و یا بخشی از کد مناسب است ، برخی اعمال همزمانی بد مانند علامت گذاری یک نخ کارگر منتظر برای شروع اجرا ، نیز در این دستور وجود دارد.

توابع API ویندوز (Win32 API) مجموعه ی غنی تری از این ساختارهای همزمان کننده را دارا می باشد ، که این ساختارها در .NET Framework با استفاده از کلاس های EventWaitHandle ، Mutex و Semphore فراهم شده است. برخی از این ساختارها نسبت به دیگری برتری هایی دارند : برای مثال کلاس Mutex ، بیشتر اوقات کدی که به وسیله دستور lock فراهم شده بود را دو برابر می کند ، در حالی که EventWaitHandle یک علامت گذاری یکتا را برای آن ایجاد می کند.

همه این کلاس ها از کلاس WaitHandle – که یک کلاس abstract است – به ارث می برند ، اگرچه از لحاظ رفتاری آنها با یکدیگر متفاوت هستند. یکی از این موارد این است که همه ی آنها ، به طور اختیاری ، می توانند نام گذاری شوند ، که با این کار به آنها اجازه داده می شود تا در طول همه پروسه های سیستم عامل کار کنند ، ترجیحاً در طول نخ های process جاری.

EventWaitHandle دو زیر کلاس دارد :کلاس AutoResetEvent و کلاس ManualResetEvent (هیچ یک از این دو کلاس برخلاف اسمشان به رویداد ها یا نماینده ها در C# ارتباطی ندارند).هر دو کلاس همه ی تمام فعالیت خود را از کلاس پایه خودشان به ارث می برند :تنها تفاوت این دو کلاس در این است که این دو کلاس سازنده های کلاس پایه را با آرگومان های متفاوت فراخوانی می کند.

در رابطه با کارایی ، overhead ناشی از همه ی کلاس هایی که مدیریت انتظار را برعهده دارند در حوزه ی کمتر از میکروثانیه قرار دارد.

AutoResetEvent پر استفاده ترین کلاس در میان کلاس های WaitHandle می باشد و به همراه ساختار lock ، یک ساختار همزمان کننده ی کلیدی است.

AutoResetEvent

یک AutoResetEvent شبیه یک دستگاه کنترل ورود عمل می کند است :با وارد کردن یک بلیط دقیقاً یک نفر اجازه عبور دارد .عبارت auto در نام کلاس به این اشاره دارد که اگر یک نفر وارد شود در آن صورت دستگاه به طور اتوماتیک reset شده و منتظر بلیط بعدی می ماند .یک نخ منتظر می ماند و یا مسدود می شود ، در مثال در ورودی ، با فراخوانی متد WaitOne (به عبارتی می گوییم که ، در پشت این در منتظر بمان تا زمانی که آن باز شود) و بلیط با فراخوانی متد Set وارد می شود. اگر تعدادی از نخ ها متد Waitone را فراخوانی کنند ، صفی در پشت این در اتوماتیک ایجاد می شود. یک بلیط می تواند از هر نخی بیاید – به عبارت دیگر ، هر نخ (مسدود نشده) با دستیابی به کلاس AutoResetEvent و فراخوانی متد Set می تواند یک نخ مسدود شده را آزاد کند.

اگر متد Set در صورتی که نخی در حالت انتظار نباشد فراخوانی شود ، handle تا زمانی که نخی متد WaitOne را فراخوانی نکند باز می ماند. این کار از رقابتی (race)که بین یک نخ برای سوراخ کردن بلیط و دیگری برای وارد کردن بلیط به دستگاه سوراخ کن ،جلوگیری می کند.(ورود بلیط کمتر از چند میکروثانیه طول خواهد کشید و خبر بد این است که شما هم اکنون می بایست برای مدت نامحدودی منتظر بمانید!) در هر صورت فراخوانی متد Set در مثال بالا ، زمانی که هیچ نخی در حالت انتظار نیست ، وقتی همه ی آنها فرا رسیدند ، به تمام بخش ها این اجازه را نمی دهد :فقط تنها فرد بعدی اجازه وارد کردن بلیط خود را دارد و بلیط های دیگر هدر خواهند رفت.

WaitOne دارای یک پارامتر timeout است –اگر انتظار به پایان برسد متد مقدار false را بازمی گرداند زیرا timeout ترجیحاً از علامت استفاده می کند .از WaitOne برای خروج از همزمان کننده ی حاضر به دلیل مدت انتظار ، البته به منظور جلوگیری از انسداد های زیاد ، نیز می تواند استفاده کرد(اگر از یک قفل گذاری اتوماتیک استفاده کنیم).

متد Reset همچنین تضمین می کند که درهای بسته شده بدون هیچ انتظار یا انسدادی ، دوباره باز خواهند شد.

یک AutoResetEvent به دو طریق می تواند ایجاد شود اولین راه به وسیله ی سازنده ی آن است :

```
EventWaitHandle wh = new AutoResetEvent (false);
```

اگر متغیر بولین دارای مقدار true باشد ، متد Set به طور اتوماتیک بعد از تولید شی ، فراخوانی می شود. راه دیگر برای تولید این شی استفاده از کلاس پایه آن است ، EventWaitHandle :

```
EventWaitHandle wh = new EventWaitHandle (false, EventResetMode.Auto);
```

سازنده ی کلاس EventWaitHandle اجازه تولید کلاس ManualResetEvent را نیز می دهد (با تعیین EventResetMode.Manual).

وقتی که نیاز به یک مدیر انتظار بعد از مدتی بر طرف گردید ، برای آزاد کردن منابع سیستم عامل می توان از متد Close استفاده کرد.

در مثال زیر ، یک نخ شروع به کار می کند ، در حالی که وظیفه ی ساده ی انتظار برای علامت دادن یک نخ دیگر را دارد ، نشان داده شده است.

```
class BasicWaitHandle
{
  static EventWaitHandle wh = new AutoResetEvent (false);

  static void Main()
  {
    new Thread (Waiter).Start();
    Thread.Sleep (1000);              // Wait for some time...
    wh.Set();                         // OK - wake it up
  }
  static void Waiter()
  {
```

```
        Console.WriteLine ("Waiting...");
        wh.WaitOne();                   // Wait for notification
        Console.WriteLine ("Notified");
    }
}
```

خروجی کد به صورت زیر است :

```
Waiting... (pause) Notified.
```

در واقع بعد از اجرای متد WaitOne ، اجرای برنامه به تابع Main بازگشته و سپس برنامه با اجرای متد Set علامت مورد نظر متد WaitOne را ایجاد کرده و کنترل دوباره به متد Waiter بازمی گردد.

تولید یک EventWaitHandle با پردازش موازی

سازنده ی این کلاس همچنین اجازه تولید یک EventWaitHandle نام گذاری شده را نیز می دهد –توانایی پردازش طولی پردازش های متعدد. نام یک رشته ی ساده است –و می تواند هر مقداری که با دیگر مقادیر ناسازگار نباشد را داشته باشد! اگر این نام هم اکنون در کامپیوتر مورد استفاده قرار گرفته باشد ، یک مرجع به همان شی EventWaitHandle با این نام ، تولید می شود ، وگرنه سیستم عامل یک نمونه جدید از این کلاس را ایجاد می کند. در زیر مثالی را ملاحظه می کنید :

```
EventWaitHandle wh = new EventWaitHandle (false, EventResetMode.Auto,
 "MyCompany.MyApp.SomeName");
```

اگر دو برنامه ی جدا این کد را اجرا کنند ، آنها قادر خواهند بود تا یکدیگر علامت گذاری کنند :مدیر انتظار در طول همه ی نخ ها در هر دو process کار خواهد کرد.

تصدیق

فرض کنید می خواهیم کارهای را در پس زمینه انجام دهیم البته بدون overhead مربوط به ایجاد یک اشیا جدید در هر بار فراخوانی این اعمال. ما این کار را می توانیم با یک نخ کارگر که متناوباً در یک حلقه است ، انجام دهیم –انتظار برای یک وظیفه ، انجام آن کار، و سپس انتظار برای کار بعدی. این کار یک سناریو چند نخی رایج است که علاوه بر قطع overhead در تولید نخ ها ، اجرای کارها نیز علامت گذاری می شود و تقابل های نا خواسته بین نخ های متعدد و افراط مصرف منابع سیستمی نیز حذف می شود.

در هر حال ، اگر نخ های ما هم اکنون در حال انجام کارهای قبلی باشند ، ما باید تصمیم بگیریم که وقتی کاری جدید می آید ، چه کاری باید انجام دهیم. فرض کنید در این حالت ما تصمیم می گیریم تا تمام اعمال را مسدود کنیم تا کارهای قبلی کاملاً انجام شود. چنین سیستمی می تواند با تولید دو شی AutoResetEvent پیاده سازی شود : یک شی AutoResetEvent که وقتی آن نخ آماده بود به وسیله ی نخ تنظیم می شود و نام آن ready می باشد و شی دیگر با نام go که به وسیله ی نخ فراخواننده وقتی که وظیفه ی جدیدی ایجاد شده است ، تنظیم می شود. در مثال زیر ، یک فیلد رشته ای ساده برای بیان نوع کار استفاده می شود (به این دلیل که تضمین کنیم که هر دو نخ همیشه همان ورژن از کار را خواهند دید این فیلد را با کلمه کلیدی volatile اعلان کرده ایم) :

```
class AcknowledgedWaitHandle
{
  static EventWaitHandle ready = new AutoResetEvent (false);
  static EventWaitHandle go = new AutoResetEvent (false);
  static volatile string task;

  static void Main()
```

```
  {
    new Thread (Work).Start();

    // Signal the worker 5 times
    for (int i = 1; i <= 5; i++)
    {
      ready.WaitOne();              // First wait until worker is ready
      task = "a".PadRight (i, 'h');  // Assign a task
      go.Set();                     // Tell worker to go!
    }

    // Tell the worker to end using a null-task
    ready.WaitOne();
    task = null;
    go.Set();
  }

  static void Work()
  {
    while (true)
    {
      ready.Set();                      // Indicate that we're ready
      go.WaitOne();                     // Wait to be kicked off...
      if (task == null)
        return;                         // Gracefully exit
      Console.WriteLine (task);
    }
  }
}
```

خروجی کد بالا به صورت زیر است :

```
ah
ahh
ahhh
ahhhh
```

ملاحظه می کنید که ما مقدار null را برای علامت دادن به نخ برای خروج ، به فیلد task منتسب کردیم. در این موقعیت فراخوانی متد های Interrupt یا Abort در نخ کارگر ، کاملاً خوب کار خواهد کرد –با فراهم آوردن این موضوع که ما ابتدا ready.WaitOne را فراخوانی کرده ایم. به این دلیل که ما بعد از فراخوانی متد ready.WaitOne در ابتدا ، ما می توانیم مکان قرارگیری کارگر را تعیین کنیم – چه در همان لحظه چه قبل از فراخوانی متد go.WaitOne –و به این وسیله از پیچیدگی های توقف دلخواهانه ی کد جلوگیری می کنیم. فراخوانی متد های Interrupt و Abort همچنین نیازمند این است که ما خطا های مهم برنامه را در نخ گارگر بگیریم.

صف تولید کننده / مصرف کننده

از دیگر سناریو های نخ داشتن یک نخ پس زمینه است که وظایف پردازش داده ها را از یک صف دریافت می کند. این صف ، صف تولید کننده / مصرف کننده نامیده می شود : تولید کننده وظایف را وارد صف می کند ؛ مصرف کننده وظایف را از آن خارج می کند و به یک نخ می سپارد. این کار تقریباً شبیه مثال قبلی ماست ، با این تفاوت که اگر نخ هم اکنون مشغول به انجام وظیفه ای باشد ، فراخواننده ، مسدود نمی شود.

یک صف تولید کننده / مصرف کننده قابل مقایسه است ، به این صورت که مصرف کننده های متعددی می تواند ایجاد شوند –هر کدام همان صف را سرویس می کند ، البته در یک نخ جدا. استفاده از فایده های سیستم های چند پردازنده ای برای محدود کردن تعداد نخ ها به منظور اجتناب از قرار گرفتن در دام نخ های همزمانی که bound نشده اند ، می تواند راه مناسبی باشد (سوییچ کردن های زیاد در میان داده ها و رقابت بین منابع سیستم).

در مثال زیر ، یک AutoResetEvent برای علامت دهی به یک نخ استفاده شده است ، و زمانی که این نخ از انجام وظایف خسته شود ، به حال انتظار در می آید (وقتی صف خالی است). یک کلاس هوشمند برای صف وجود دارد ، که دستیابی به آن باید توسط یک بلوک lock حفظ شود تا مسئله ی امنیت نخ را تضمین کند. کار نخ با ورود وظیفه ی null به صف پایان می پذیرد :

```
using System;
using System.Threading;
using System.Collections.Generic;

class ProducerConsumerQueue : IDisposable
{
  EventWaitHandle wh = new AutoResetEvent (false);
  Thread worker;
  object locker = new object();
  Queue<string> tasks = new Queue<string>();

  public ProducerConsumerQueue()
  {
    worker = new Thread (Work);
    worker.Start();
  }

  public void EnqueueTask (string task)
  {
    lock (locker)
      tasks.Enqueue (task);
    wh.Set();
  }

  public void Dispose()
  {
    EnqueueTask (null);     // Signal the consumer to exit.
    worker.Join();          // Wait for the consumer's thread to finish.
    wh.Close();             // Release any OS resources.
  }

  void Work()
  {
    while (true)
    {
      string task = null;
      lock (locker)
      if (tasks.Count > 0)
      {
        task = tasks.Dequeue();
        if (task == null)
          return;
      }
      if (task != null)
      {
        Console.WriteLine ("Performing task: " + task);
        Thread.Sleep (1000);  // simulate work...
      }
      else
        wh.WaitOne();         // No more tasks - wait for a signal
    }
  }
}
```

در زیر متد Main برای آزمایش این کد آورده شده است :

```
class Test
{
  static void Main()
  {
    using (ProducerConsumerQueue q = new ProducerConsumerQueue())
    {
      q.EnqueueTask ("Hello");
      for (int i = 0; i < 10; i++) q.EnqueueTask ("Say " + i);
      q.EnqueueTask ("Goodbye!");
    }
    // Exiting the using statement calls q's Dispose method, which
    // enqueues a null task and waits until the consumer finishes.
  }
}
```

در زیر خروجی این کد آمده است :

```
Performing task: Hello
Performing task: Say 1
Performing task: Say 2
Performing task: Say 3
...
...
Performing task: Say 9
Goodbye!
```

توجه کنید که در این مثال وقتی که شی ProducerConsumerQueue نابود می شود ، ما به طور غیر مستقیم از مدیر انتظار خارج شده ایم –با این حال ما می توانیم ذاتاً نمونه های بسیاری از این کلاس را در طول برنامه ایجاد کرده و از بین ببریم.

ManualResetEvent

یک ManualResetEvent نوعی از AutoResetEvent می باشد. تفاوت آن در این است که ، نمی تواند به طور اتوماتیک بعد از اینکه یک نخ اجازه حرکت به وسیله ی متد WaitOne را پیدا کرد ، خود را reset کند و بنابراین مانند یک گیت عمل می کند: فراخوانی Set این دروازه را باز می کند ، به هر تعداد نخ که در جلوی گیت قرار دارند اجازه ی WaitOne را می دهد ؛ فراخوانی Close گیت را می بندد ، و صفی از منتظران تا گشایش بعدی گیت جمع آوری می کند.

می توان این کارکرد را به وسیله ی یک فیلد بولین با نام gateopen شبیه سازی کرد.(البته با کلمه کلیدی volatile باید اعلان شود) در ترکیب با spin-sleeping –کراراً این پرچم چک شده ، و سپس برای مدت کمی این نخ به خواب رود.

ManualResetEvent معمولاً برای علامت گذاری هنگام پایان یک عمل مشخص استفاده می شود ، یا اینکه نشان دهد مقداردهی اولیه به نخ کامل شده است و برای انجام کارها آماده است .

Mutex

Mutex همان کارایی که دستور lock ارائه می داد را فراهم می کند ، Mutex افزونگی را بیشتر می کند. یکی از فواید این کلاس کار کردن در میان process های متعدد می باشد. یک قفل که در طول کامپیوتر فعالیت می کند را در مقابل یک قفل که در طول برنامه فعالیت می کند در اختیار ما قرار می دهد.

با اینکه Mutex منطقاً سریع می باشد ، اما دستور lock هزاران بار سریعتر از آن است. دستیابی به Mutex چند میکروثانیه زمان می برد؛ در حالی که دستیابی به یک دستور lock ده ها نانوثانیه زمان خواهد برد (با فرض هیچ انسدادی)

با یک کلاس Mutex ، متد WaitOne یک قفل انحصاری را فراهم می آورد ، در صورت مخالفت این نخ مسدود می شود. قفل انحصاری سپس با متد ReleaseMutex آزاد می شود. مانند دستور lock در C# یک Mutex فقط از همان نخی که آزاد شده است می تواند احراز شود.

یک استفاده رایج برای یک Mutex دارای پردازش موازی ، این است که تضمین می کند یک نمونه از برنامه می تواند در یک زمان اجرا شود. در زیر طریقه این پیاده سازی آمده است :

```
class OneAtATimePlease
{
  // Use a name unique to the application (eg include your company URL)
  static Mutex mutex = new Mutex (false, "oreilly.com OneAtATimeDemo");

  static void Main()
  {
    // Wait 5 seconds if contended – in case another instance
    // of the program is in the process of shutting down.

    if (!mutex.WaitOne (TimeSpan.FromSeconds (5), false))
    {
      Console.WriteLine ("Another instance of the app is running. Bye!");
      return;
    }
    try
    {
      Console.WriteLine ("Running - press Enter to exit");
      Console.ReadLine();
    }
     finally
        {
           mutex.ReleaseMutex();
        }
  }
}
```

یکی از ویژگی های خوب Mutex این است که اگر برنامه ای بدون فراخوانی ReleaseMutex ناگهان پایان یابد ، CLR شی Mutex را به طور اتوماتیک آزاد می کند.

Semaphore

یک Semaphore مانند یک کلوپ شبانه روزی است : دارای ظرفیت معینی است. هنگامی که کلوپ پر شد فرد دیگری نمی تواند وارد کلوپ شود و یک صفی از افراد در بیرون کلوپ تولید می شود. سپس به ازای هر فردی که خارج می شود یک نفر از ابتدای صف می تواند وارد شود.سازنده ی کلاس حداقل نیازمند دو آرگومان می باشد –تعداد مکان هایی که در حال حاضر در این کلوپ شبانه روزی در دسترس است و کل ظرفیت کلوپ شبانه روزی.

یک semaphore با ظرفیت یک شبیه یک Mutex یا lock رفتار می کند ، به استثنای اینکه یک Semaphore هیچ مالکی ندارد –این شی یک *thread-agnostic* است. هر نخ می تواند متد Release را در یک Semaphore فراخوانی کند ، مادامی که با Mutex و lock ، فقط نخی که از منبع سیستم استفاده کرده است می تواند آنرا آزاد کند.

در مثال زیر ، ده نخ یک حلقه را با دستور Sleep ایجاد می کند. یک Semaphore تضمین می کند که بیشتر از سه نخ نتواند متد sleep را یک دفعه با هم اجرا کنند :

```
class SemaphoreTest
{
  static Semaphore s = new Semaphore (3, 3);  // Available=3; Capacity=3

  static void Main()
  {
    for (int i = 0; i < 10; i++)
      new Thread (Go).Start();
  }

  static void Go()
  {
    while (true)
    {
      s.WaitOne();
      Thread.Sleep (100);   // Only 3 threads can get here at once
      s.Release();
    }
  }
}
```

SignalAndWait و WaitAll ، WaitAny

علاوه بر متد های Set و WaitOne ، متد های استاتیکی در کلاس WaitHandle وجود دارد تا جلوه های پیچیده تری از ابزار های همزمان کننده را برای ما ارائه کند.

متد های WaitAny ، WaitAll و SignalAndWait انتظار برای مدیران انتظار را آسان تر کرده اند.

SignalAndWait شاید مفید ترین این متد ها باشد !این متد –در یک عمل atomic ، متد WaitOne در یک WaitHandle فراخوانی می کند ، مادامی که متد Set را در WaitHandle دیگر فراخوانی می کند. برای تنظیم اینکه دو نخ در یک مدل کتابی ، در یک زمان بتوانند یکدیگر را ملاقات کنند می توان از یکی از EventWaitHandle ها استفاده کرد. هم AutoResetEvent و ManualResetEvent این کار را انجام می دهند.
اولین نخ این کار را انجام می دهد :

```
WaitHandle.SignalAndWait ( wh1 ,  wh2 );
```

در حالی که نخ دوم مخالف این کار را انجام می دهد :

```
WaitHandle.SignalAndWait (wh2, wh1);
```

WaitHandle.WaitAny منتظرهر یک از عناصر آرایه ای از مدیران انتظار می ماند؛ WaitHandle.WaitAll برای تمام handler ها منتظر می ماند .مشابه استفاده از دستگاه کنترل ورود .این متد ها در یک زمان شبیه صفبندی در تمام دستگاه های ورود هستند –حرکت برای بازشدن در اولین فراخوانی (در این موقعیت WaitAny) یا انتظار برای بازشدن همه آنها (در این موقعیت WaitAll)

WaitAll به خاطر ارتباط عجیبی که با apartment threading دارد دارای ارزش مشکوکی است –بازگشتی از میراث معماری WaitAll. COM نیازمند این است که فراخواننده در یک ساختار چند نخی باشد –که مدلی مناسب برای اعمال مقایسه ای می باشد .به ویژه برای برنامه های تحت ویندوز ، که نیاز دارند تا کارهای خود را طبیعی تر از ارتباط آن با حافظه کلیپ برد انجام دهند!

خوشبختانه NET Framework. ابزار پیشرفته تری را برای زمانی که ابزارهای انتظار نامناسب هستند ، نیز فراهم کرده است – Monitor.Wait و Monitor.Pulse.

بافت همزمان کننده (Synchronizing Context)

نسبت به قفل گذاری دستی ، می توان قفل گذاری را به صورت اعلانی نیز انجام داد. با اشتقاق از کلاس ContextBoundObject و قرار دادن صفت Synchronization ، CLR را طوری شکل می دهیم که به طور اتوماتیک عمل قفل گذاری را انجام دهد. در زیر مثالی آورده شده است :

```
using System;
using System.Threading;
using System.Runtime.Remoting.Contexts;

[Synchronization]
public class AutoLock : ContextBoundObject
{
  public void Demo()
  {
    Console.Write ("Start...");
    Thread.Sleep (1000);          // We can't be preempted here
    Console.WriteLine ("end");    // thanks to automatic locking!
  }
}

public class Test
{
  public static void Main()
  {
    AutoLock safeInstance = new AutoLock();
    new Thread (safeInstance.Demo).Start();    // Call the Demo
    new Thread (safeInstance.Demo).Start();    // method 3 times
    safeInstance.Demo();                       // concurrently.
  }
}
```

خروجی کد به صورت زیر است :

```
Start... end
Start... end
Start... end
```

CLR تضمین می کند که فقط یک نخ می تواند کد را در SafeInstance و در یک زمان اجرا کند .او این کار را با ایجاد یک شی همزمان کننده انجام می دهد –و آنرا حول هر فراخوانی برای هر متد یا صفت SafeInstance قفل گذاری می کند .حوزه ی قفل – در این موقعیت – شی SafeInstance – یک بافت همزمان کننده (*Synchronizing Context*) نامیده می شود.

حال ، این شی چه طور کار می کند؟ یک راهنما در فضای نام صفت Synchronization قرار دارد : System.Runtime.Remoting.Contexts. یک شی ContextBoundObject می تواند به عنوان یک شی از راه دور دیده شود –به این معنی که همه ی فراخوانی متدها با هم تقاطع دارند. برای اینکه این تقاطع را ممکن کنیم ، وقتی که ما AutoLock را ایجاد می کنیم ، CLR در واقع یک Proxy باز می گرداند – شی ای با همان متد ها و صفات یک شی AutoLock ، که به عنوان یک واسطه عمل می کند. رخ دادن اتوماتیک قفل گذاری به خاطر همین عنصر واسطه است. روی هم رفته عمل تقاطع ، حول وحوش یک میکروثانیه به فراخوانی هر متد اضافه می کند.

همزمانی اتوماتیک برای حفاظت از اعضای استاتیک یا کلاس هایی که از کلاس ContextBoundObject مشتق نشده باشند (برای مثال یک فرم ویندوز) نمی تواند استفاده شود.

قفل گذاری به همان صورت قبلی انجام می گیرد. شما احتمالاً انتظار دارید که کد زیر همان نتیجه قبلی را تولید کند :

```
[Synchronization]
```

```
public class AutoLock : ContextBoundObject
{
   public void Demo()
   {
     Console.Write ("Start...");
     Thread.Sleep (1000);
     Console.WriteLine ("end");
   }

   public void Test()
   {
     new Thread (Demo).Start();
     new Thread (Demo).Start();
     new Thread (Demo).Start();
     Console.ReadLine();
   }

   public static void Main()
   {
     new AutoLock().Test();
   }
}
```

(ملاحظه می کنید که ما در دستور Console.ReadLine خود را پنهان کرده ایم). زیرا فقط یک نخ می تواند کد را در یک زمان نشان دهد ، سه نخ دیگر تا زمانی که متد Test پایان یابد ، در متد Demo به حالت مسدود باقی می ماند –که برای کامل شدن نیازمند دستور ReadLine می باشد. از این رو بعد از زدن کلید Enter ، ما با همان نتیجه ی قبلی رو به رو خواهیم بود.این کد یکی از آسیب های امنیت نخ است و تقریباً یکی از بزرگترین ضربات آن بشمار می رود، زیرا از هر فایده ای که چند خطی در کلاس تولید می کند ، پیش گیری می کند!

بنابراین ، ما مشکلی که قبلاً ارائه شد را حل نکردیم : اگر AutoLock یک کلاس مجموعه باشد ، برای نمونه ، ما هنوز نیازمند یک قفل حول یک دستور مانند زیر هستیم ، فرض کنید این کد در کلاس دیگری اجرا خواهد شد:

```
if ( safeInstance.Count > 0 ) safeInstance.RemoveAt (0) ;
```

مگر اینکه کلاس کد بالا خودش یک ContextBoundObject همزمان شده باشد.

یک بافت همزمانی می تواند ماورای حوزه ی یک شی بسط داده شود. به طور پیش فرض اگر یک شی همزمان شده از درون کد دیگری معرفی شود ، هر دوی این کارها همان بافت را به اشتراک می گذارد (به عبارت دیگر ، یک قفل بزرگ) این رفتار می تواند با تعیین یک پرچم از اعداد صحیح در سازنده ی صفت Synchronization ، تغییر کند ، با استفاده از یکی از ثوابت موجود در کلاس SynchronizationAttribute :

نام ثابت	مفهوم
NOT_SUPPORTED	مساوی با استفاده نکردن از صفت synchronized است
SUPPORTED	اگر بافت همزمانی حاضر از دیگر اشیای همزمان شده معرفی شود ، آنها را با هم ترکیب می کند ، وگرنه همچنان هماهنگ نشده باقی می ماند
REQUIRED(default)	اگر بافت همزمانی حاضر از دیگر اشیای همزمان شده معرفی شود ، آنها را با هم ترکیب می کند ، وگرنه یک بافت جدید ایجاد می کند
REQUIRES_NEW	همیشه یک بافت همزمان کننده ی جدید ایجاد می کند

بنابراین اگر شی کلاس *SynchronizedA* توسط شی دیگری از کلاس *SynchronizedB* معرفی شود ، اگر *SynchronizedB* به روش زیر اعلان شود ، آنها بافت همزمانی جداگانه ای می گیرند :

```
[Synchronization (SynchronizationAttribute.REQUIRES_NEW)]
public class SynchronizedB : ContextBoundObject { ...
```

حوزه ی بافت همزمانی بیشتر مدیریت آنرا آسان تر می کند ، اما فرصت برای یک همزمانی مفید را کم می کند. در مقیاس پایانی ، بافت های همزمان کننده ی جدا ، بن بست ها را تولید می کنند. به مثال زیر توجه کنید:

```
[Synchronization]
public class Deadlock : ContextBoundObject
{
  public DeadLock Other;
  public void Demo()
  {
    Thread.Sleep (1000);
    Other.Hello();
  }
  void Hello()
  {
    Console.WriteLine ("hello");
  }
}

public class Test
{
  static void Main()
  {
    Deadlock dead1 = new Deadlock();
    Deadlock dead2 = new Deadlock();
    dead1.Other = dead2;
    dead2.Other = dead1;
    new Thread (dead1.Demo).Start();
    dead2.Demo();
  }
}
```

زیرا هر نمونه از Deadlock درون Test ایجاد شده است –یک کلاس هماهنگ نشده –هر نمونه بافت همزمانی خودش را می گیرد ، و بنابراین ، قفل خودش را دارد. وقتی دو شی یکدیگر را فرا می خوانند ، برای تولید بن بست زمان زیاد طول نخواهد کشید ! مشکل به ویژه زمانی زیاد می شود که کلاس های Deadlock و Test توسط دو تیم برنامه نویسی نوشته شده باشد. ممکن است غیر منطقی به نظر برسد که از کلاس Test انتظار داشته باشیم تا از تخلف خود آگاه باشد ، بیایید راه حل این مشکل را با هم بررسی کنیم. این موضوع مخالف با قفل های انحصاری است ، که بن بست ها را معمولاً بیشتر نشان می دهند.

ورود مجدد (Reentrancy)

یک متد thread-safe قالباً *reentrant* نامیده می شود ، زیرا این متد می تواند در طول اجرا خود ، از بخشی از مسیر پیشدستی کند ، و سپس دوباره در نخ دیگر بدون هیچ مشکلی فراخوانی شود. در یک دید کلی ، لغات thread-safe و reentrant مترادف یا بسیار نزدیک به هم در نظر گرفته می شوند.

Reentrancy در هر حال ، دلایل نادرست خود را در قفل کردن اتوماتیک دارد .اگر صفت Synchronizing با یک آرگومان (دارای مقدار true) ورودی (reentrant) همراه باشد :

```
[Synchronization(true)]
```

سپس قفل بافت هماهنگ کننده وقتی اجرا از این بافت خارج می شود ، موقتاً آزاد خواهد شد. در مثال قبلی ، این کار از تولید بن بست ها جلوگیری می کند ؛ بدیهی است که این کار امری مطلوب می باشد. در هر حال ، اثر بدی در این خلال ایجاد می شود ، هر نخی آزاد است تا هر متدی از شی اصلی را فراخوانی کند (ورود دوباره بافت هماهنگ کننده) و از پیچیدگی های بسیار چند نخی شما را رها خواهد کرد. این است مشکل *reentrancy*

با اینکه reentrancy می تواند خطرناک باشد ، بعضی موقع تنظیمات کوچک دیگری وجود دارند. برای مثال ، فرض کنید کسی می بایست چند نخی را درون یک کلاس هماهنگ کننده پیاده سازی کند ، البته به وسیله ی نماینده کردن (delegate) منطق کار ، برای نخ هایی که اشیا را در بافت های جدا اجرا می کنند. این نخ ممکن است بدون دلیل مانع از ارتباط با دیگر نخ ها یا شی اصلی بدون استفاده از reentrancy شود.

این موضوع ضعف اساسی استفاده از همزمانی اتوماتیک را نشان می دهد : حدود زیاد در مورد اینکه چه قفل گذاری ای بکار برده می شود ، می تواند واقعاً دشواری را تولید کند که هرگز رخ نخواهد داد. این دشواری ها – بن بست ، reentrancy و سست کردن همزمانی –می تواند قفل گذاری دستی را غلاوه بر نمونه های ساده مطبوع تر می کند.

فصل سوم: استفاده از نخ ها

آپارتمان ها (Apartment) و فرم های ویندوزی

Apartment threading یک روش امنیت نخ اتوماتیک می باشد ، که خیلی نزدیک با سیستم COM-Microsoft's legacy Component Object Model پیوند خورده است. مادامی که .NET. به طور وسیعی آزادی مدل های نخ کشی (threading) را نقض کرده است ، زمان هایی نیز وجود دارد که به طور ناخود آگاهی و به دلیل نیاز به کار با API های قدیمی ، این اتفاق می افتد. نخ کشی آپارتمان (apartment threading) بیشتر مربوط به برنامه های دارای فرم می باشد ، زیرا بیشتر فرم های ویندوز از توابع API مربوط به Win32 استفاده می کنند – با میراث آپارتمان مربوطه اش کامل می شود.

یک آپارتمان یک کانتینر منطقی برای نخ هاست. آپارتمان ها در دو اندازه موجودند – تکی و چندتایی. مدل تکی فقط شامل یک نخ می باشد ؛ در حالی که در مدل چند تایی هر تعدادی از نخ های می تواند قرار بگیرد. مدل تکی از مدل دیگر رایج تر و کاراتر می باشد.

به علاوه ی قرار داشتن نخ ها ، آپارتمان ها شامل اشیا نیز می باشند. وقتی شی ای درون آپارتمانی ایجاد می شود ، تمام طول عمر خود را در آن قرار می گیرد و برای همیشه همسایه ی نخ ها خواهد بود. این کار شبیه شی ای است که در .NET Synchronization Context. (که در بخش دو توضیح داده شد)وجود دارد ، به جز اینکه یک وضعیت همزمان کننده شامل نخ ها نمی باشد. هر نخ می تواند یک شی را در هر وضعیت همزمان کننده فراخوانی کند – به موضوع انتظار برای یک قفل انحصاری اشاره دارد. اما اشیا درون یک آپارتمان فقط به وسیله نخ های درون آپارتمان می توانند فراخوانی شوند.

کتابخانه ای را تصور کنید که هر کتاب در آن حکم یک شی را دارد. قرض گرفتن کتاب اجازه داده نمی شود – کتاب ها در کتابخانه برای تمام عمر همان جا باقی می مانند. در نهایت ، یک شخص را به عنوان یک نخ فرض کنید.

یک کتابخانه ی وضعیت همزمان کننده به هر شخص اجازه ی ورود می دهد ، همچنین در هر زمان فقط یک نفر وارد می شود ، و صفی از افراد در بیرون کتابخانه شکل می گیرد.

یک کتابخانه آپارتمان دارای پرسنل مستقر در کتابخانه می باشد – یک کتابدار برای یک کتابخانه ی تک نخه ، و بقیه تیم برای یک کتابخانه ی چند نخه. هیچ کسی به جز مسئولان اجازه ی ورود به این کتابخانه را ندارند – مشتری که می خواهد تحقیقی را انجام دهد باید از طرف کتابدار علامتی را دریافت کند

، سپس از کتابدار بپرسد که آیا کار را انجام دهد یا خیر ! علامتی که کتابدار نشان می دهد marshalling نامیده می شود – مشتری فراخوانی متد را به سمت عضو پرسنلی هدایت می کند ، عمل marshalling اتوماتیک انجام می شود و در سمت کتابدار با فرستادن پیغامی پیاده سازی می شود – در فرم های ویندوز ، این مکانیزمی است که برای دریافت رویداد های صفحه کلید یا ماوس توسط سیستم عامل انجام می گیرد. اگر پیام ها به سرعت برای پردازش برسند ، آنها وارد یک صف پیام می شوند ، بنابراین آنها به ترتیبی که می رسند ، می توانند پردازش شوند.

توصیف یک مدل آپارتمان

یک نخ .NET. به طور اتوماتیک و به محض ورود آپارتمان Win32 یا کد COM به یک آپارتمان انتساب داده می شود. ، این نخ به طور پیش فرض برای یک آپارتمان چند نخه حافظه در نظر می گیرد ، مگر اینکه درخواستی برای یک آپارتمان تک نخه مانند زیر دریافت شود :

```
Thread t = new Thread (...);
t.SetApartmentState (ApartmentState.STA);
```

همچنین با استفاده از صفت STAThread در متد Main می توان درخواست اتصال نخ اصلی برنامه به یک آپارتمان تک نخه را داد:

```
class Program {
  [STAThread]
  static void Main() {
  ...
```

آپارتمان ها هیچ اثری تا زمان اجرای کدهای .NET. ندارند. به عبارت دیگر ، دو نخ با یک حالت آپارتمان STA می تواند به طور همزمان از همان متد و همان شی فراخوانی شوند ، و هیچ marshalling یا قفل گذاری اتوماتیکی رخ نخواهد داد. فقط وقتی اجرای برنامه به کد مدیریت نشده می رسد می تواند این اعمال رخ دهد.

انواع داده موجود در فضای نام System.Windows.Forms به طور ممتد کدهای Win32 که برای کار در یک آپارتمان تک نخه طراحی شده اند ، را فراخوانی می کند. به این دلیل ، یک برنامه ویندوزی باید صفت STAThread را در متد Main داشته باشد ، وگرنه یکی از دو مورد زیر در کد Win32 UI اتفاق می افتد :

- کد به یک آپارتمان تک نخه راهنمایی می شود.
- برنامه crash خواهد کرد.

Control.Invoke

در برنامه های ویندوزی چند نخه ، فراخوانی متد یا ویژگی ای (property) برای یک کنترل از هر نخی که آنرا ایجاد نکرده است ، غیر منطقی می باشد. همه ی فراخوانی های cross-thread ، با استفاده از متد Control.Invoke یا Control.BeginInvoke ، باید مستقیماً به نخی که کنترل را ایجاد کرده است (معمولاً نخ اصلی) هدایت شوند. این نخ نمی تواند به marshalling اتوماتیک استناد کند زیرا بسیار دیر رخ می دهد – فقط وقتی اجرای برنامه به کد مدیریت نشده برسد ، در این زمان بسیاری از کد های داخلی .NET. ممکن است در یک نخ نادرست اجرا شود – کدی که thread-safe نیست.

WPF شبیه به فرم های ویندوز است ، به این صورت که عناصر فقط از نخی که آنرا ایجاد کرده است قابل دسترسی می باشند. متد متناسب با Control.Invoke در WPF متد Dispatcher.Invoke می باشد.

یک راه حل عالی در برنامه های ویندوزی و برنامه های WPF استفاده از کنترل BackgroundWorker می باشد. این کلاس نخ هایی که نیاز به گزارش پیشروی و پایان را دارند ، در اختیار دارد ، و به طور اتوماتیک متد Control.Invoke یا Dispatcher.Invoke را فراخوانی می کند.

BackgroundWorker

BackgroundWorker یک کلاس کمکی در فضای نام System.ComponentModel برای مدیریت یک نخ کارگر می باشد. این کلاس دارای ویژگی های زیر می باشد :

- پرچم cancel برای علامت دهی به یک نخ به منظور پایان کار خود بدون فراخوانی متد Abort
- یک پروتکل استاندارد برای گزارش پیشروی ، پایان و کنسل
- پیاده سازی اینترفیس IComponent که به این کلاس اجازه داده است تا در محیط طراح ویژوال استودیو قرار گیرد
- مدیریت خطا در نخ کارگر
- توانایی برای بروزرسانی فرم های ویندوز و کنترل های WPF در پاسخ به پیشروی یا پایان یک نخ

دو ویژگی آخر به طور ویژه ای مفید هستند – به این معنی که شما نیاز به قرار دادن هیچ بلوک try/catch ای در متد کارگر خودتان نیستید ، و فرم های ویندوز و کنترل های WPF را می تواند بدون احتیاج به فراخوانی متد Control.Invoke بروزرسانی کند.

BackgroundWorker از thread-pool (در همین بخش توضیح داده می شود) استفاده می کند ، که نخ ها را به منظور ایجاد نشدن دوباره به ازای تولید وظایف جدید بازیابی می کند. این کار به این معنی است که ما هرگز نباید متد Abort را از یک نخ BackgroundWorker فراخوانی کنیم.

در زیر کمترین مراحل برای استفاده از BackgroundWorker قرار داده شده است :

- یک شی BackgroundWorker ایجاد کنید ، و رویداد DoWork آنرا به دست بگیرید
- متد RunWorkerAsync آنرا فراخوانی کنید ، به طور اختیاری می توانید یک آرگومان از نوع object نیز به آن پاس دهید.

آرگومان پاس داده شده به متد RunWorkerAsync به رویداد DoWork پاس داده می شود ، به وسیله ویژگی Argument مربوط به آرگومان ورودی به رویداد. در زیر مثالی نشان داده شده است :

```
class Program
{
  static BackgroundWorker bw = new BackgroundWorker();
  static void Main()
  {
    bw.DoWork += bw_DoWork;
    bw.RunWorkerAsync ("Message to worker");
    Console.ReadLine();
  }

  static void bw_DoWork (object sender, DoWorkEventArgs e)
  {
    // This is called on the worker thread
    Console.WriteLine (e.Argument);        // writes "Message to worker"
    // Perform time-consuming task...
  }
```

BackgroundWorker همچنین دارای رویدادی با نام RunWorkerCompleted است و بعد از اینکه رویداد کار خود را به پایان رساند ، فراخوانی می شود. مدیریت رویداد RunWorkerCompleted اجباری نیست ، اما معمولاً به منظور جستجو در خطاهای تولید شده در رویداد DoWork استفاده می شود. علاوه بر این کد درون این رویداد می تواند فرم های ویندوز و کنترل های WPF را بدون marshalling مستقیم ، بروزرسانی کند ؛ کد درون رویداد DoWork این کار را نمی تواند انجام دهد.

برای اضافه کردن قابلیت گزارش پیش روی این کار ها را باید انجام داد :

- صفت WorkerReportsProgress را با مقدار true تنظیم می کنیم
- به طور دوره ای ، متد ReportProgress را از درون رویداد DoWork با پیغام "Percentage Complete" فراخوانی کنیم ، و به طور اختیاری می توانید یک شی مربوط به وضعیت کاربر را نیز به این متد پاس بدهید
- مدیریت رویداد ProgressChanged ، صفت ProgressPercentage مربوط به آرگومان رویداد را جستجو (query) کنید

کد درون رویداد ProgressChanged همانند رویداد RunWorkerCompleted می تواند آزادانه با کنترل های UI کار کند. معمولاً این جا جایی است که شما یک progress bar را بروزرسانی می کنید.

برای اضافه کردن قابلیت کنسل عمل شی این کار ها را باید انجام داد :

- صفت WorkerSupportsCancellation را با مقدار true تنظیم کنید
- به طور دوره ای صفت CancellationPending را از درون رویداد DoWork چک کنید – اگر دارای مقدار true بود ، صفت Cancel آرگومان ورودی رویداد را با مقدار true تنظیم کنید ، و دستور return را فراخوانی کنید. (شی BackgroundWorker می تواند مقدار صفت Cancel را true کند و بدون برانگیخته شدن توسط CancellationPending خارج شود – البته اگر تصمیم بگیرد تا کار بسیار مشکلی را انجام دهد)
- به منظور درخواست عمل لغو پردازش متد CancelAsync را فراخوانی کنید.

در زیر مثالی قرار دارد که تمام مطالب بالا را پیاده سازی کرده است :

```
using System;
using System.Threading;
using System.ComponentModel;

class Program
{
  static BackgroundWorker bw;
  static void Main()
  {
    bw = new BackgroundWorker();
    bw.WorkerReportsProgress = true;
    bw.WorkerSupportsCancellation = true;
    bw.DoWork += bw_DoWork;
    bw.ProgressChanged += bw_ProgressChanged;
    bw.RunWorkerCompleted += bw_RunWorkerCompleted;

    bw.RunWorkerAsync ("Hello to worker");
```

```
    Console.WriteLine ("Press Enter in the next 5 seconds to cancel");
    Console.ReadLine();
    if (bw.IsBusy)
      bw.CancelAsync();
    Console.ReadLine();
  }

  static void bw_DoWork (object sender, DoWorkEventArgs e)
  {
    for (int i = 0; i <= 100; i += 20)
    {
      if (bw.CancellationPending)
      {
        e.Cancel = true;
        return;
      }
      bw.ReportProgress (i);
      Thread.Sleep (1000);
    }
    e.Result = 123;    // This gets passed to RunWorkerCompleted
  }

  static void bw_RunWorkerCompleted (object sender,
  RunWorkerCompletedEventArgs e)
  {
    if (e.Cancelled)
      Console.WriteLine ("You cancelled!");
    else if (e.Error != null)
      Console.WriteLine ("Worker exception: " + e.Error.ToString());
    else
      Console.WriteLine ("Complete - " + e.Result);      // from DoWork
  }

  static void bw_ProgressChanged (object sender,
  ProgressChangedEventArgs e)
  {
    Console.WriteLine ("Reached " + e.ProgressPercentage + "%");
  }
}
```

خروجی کد بالا در زیر نمایش داده شده است :

```
Press Enter in the next 5 seconds to cancel
Reached 0%
Reached 20%
Reached 40%
Reached 60%
Reached 80%
Reached 100%
Complete – 123

Press Enter in the next 5 seconds to cancel
```

```
Reached 0%
Reached 20%
Reached 40%

You cancelled!
```

ارث بری از BackgroundWorker

BackgroundWorker یک کلاس sealed نیست و متد مجازی ای (virtual) با نام OnDoWork فراهم آورده است ، که به دیگر الگوها پیشنهاد می کند تا از آن استفاده کند. وقتی متدی که اجرای آن ذاتاً زمان زیادی طول می کشد را می نویسیم ، به جای آن – یا به همراه آن – می توانیم ورژنی از متد را بنویسیم که یک زیر کلاس از BackgroundWorker را بازمی گرداند ، که از قبل برای انجام کارها به صورت غیر همزمانی (asynchronously) پیکربندی شده است. برنامه نویس فقط نیاز دارد تا رویداد های RunWorkerCompleted و ProgressChanged را مدیریت کند. برای مثال ، ما متدی با نام GetFinancialTotals نوشته ایم که زمان زیادی را برای پردازش صرف می کند :

```
public class Client
{
  Dictionary <string,int> GetFinancialTotals (int foo, int bar) { ... }
  ...
}
```

ما می توانیم آنرا به صورت زیر بنویسیم :

```
public class Client
{
  public FinancialWorker GetFinancialTotalsBackground (int foo, int bar)
  {
    return new FinancialWorker (foo, bar);
  }
}

public class FinancialWorker : BackgroundWorker
{
  public Dictionary <string,int> Result;   // We can add typed fields.
  public volatile int Foo, Bar;          // We could even expose them
                                 // via properties with locks!
  public FinancialWorker()
  {
    WorkerReportsProgress = true;
    WorkerSupportsCancellation = true;
  }

  public FinancialWorker (int foo, int bar) : this()
  {
    this.Foo = foo; this.Bar = bar;
  }
```

```
  protected override void OnDoWork (DoWorkEventArgs e)
  {
    ReportProgress (0, "Working hard on this report...");
    Initialize financial report data

    while (!finished report )
    {
      if (CancellationPending)
      {
        e.Cancel = true;
        return;
      }
      Perform another calculation step
      ReportProgress (percentCompleteCalc, "Getting there...");
    }
    ReportProgress (100, "Done!");
    e.Result = Result = completed report data;
  }
}
```

هر کسی که متد GetFinancialTotalsBackground را فراخوانی کند یک شی FinancialWorker دریافت می کند. این شی می تواند پیشروی و لغو شدن برنامه را گزارش دهد و نیز با فرم های ویندوز بدون فراخوانی متد Control.Invoke سازگار است. این شی همچنین قابلیت مدیریت خطا را دارد و از یک پروتکل استاندارد استفاده می کند.

استفاده از BackgroundWorker به طور مؤثری استفاده از "الگوی ناهماهنگ (asynchronous) بر پایه رویداد" را منقضی کرد.

ReaderWriterLockSlim و ReaderWriterLock

تقریباً همیشه ، نمونه های یک نوع داده برای عملیات خواندن همزمان thread-safe می باشد ، اما برای بروزرسانی همزمان و همچنین خواندن و بروزرسانی همزمان اینگونه نیست. این مطلب همچنین برای منابع اطلاعاتی ای مانند فایل ها نیز درست است. اگرچه حفاظت از این نمونه ها با یک قفل انحصاری ساده برای همه حالات دستیابی معمولاً ایده ای مکارانه است ، اما اگر تعداد زیادی خواننده ی فایل و بروزرسانی های اتفاقی وجود داشته باشند ، می تواند همزمانی را بی جهت محدود کند. مثالی از مکانی که ممکن است این اتفاق بیفتد در سرور برنامه های تجاری است ، جایی که اطلاعات رایج برای دستیابی سریعتر از فیلدهای استاتیک در حافظه cache می شوند. کلاس ReaderWriterLockSlim برای ایجاد بیشترین سطح دسترسی قفل گذاری در چنین سناریو هایی طراحی شده است.

کلاس ReaderWriterLockSlim در Framework 3.5 قرار داده شده است و جایگزینی برای کلاس قدیمی ReaderWriterLock می باشد. کاری که کلاس ReaderWriterLock انجام می دهد مشابه این کلاس است ، اما بسیار کندتر از آن است و دارای اشتباه طراحی ذاتی در مکانیزم خود برای بهبود مدیریت قفل گذاری می باشد.

با هر دو کلاس دو نوع قفل پایه وجود دارد : یک قفل برای خواندن و قفلی برای نوشتن. یک قفل نوشتن عموماً انحصاری است ، اگرچه یک قفل خواندن با دیگر قفل های خواندن سازگار است.

بنابراین ، یک نخی که یک قفل نوشتن را نگه می دارد ، همه ی دیگر نخ ها را که تلاش کنند تا یک قفل خواندن یا نوشتن را احراز کنند ، مسدود می کند. اما اگر هیچ نخی یک قفل نوشتن را نگه ندارد ، هر تعدادی از نخ ها ممکن است یک قفل خواندن را به طور همزمان فراهم کند.

کلاس ReaderWriterLockSlim متد های زیر را برای فراهم آوردن/آزاد کردن قفل های خواندن/نوشتن تعریف کرده است :

- public void EnterReadLock();
- public void ExitReadLock();
- public void EnterWriteLock();
- public void ExitWriteLock();

علاوه بر این ها ، ورژن های Try همه متدهای EnterXXX وجود دارند که آرگومان های timeout را در فرم Monitor.TryEnter می پذیرند(timeout ها می توانند به راحتی هر زمان که با منبع داده ای مخالفت شد ، اتفاق بیافتند). کلاس ReaderWriterLock متدهای مشابهی با نام های AcquireXXX و ReleaseXXX فراهم می آورد. اگر timeout اتفاق بیافتد ، این متدها به جای بازگرداندن مقدار false یک خطای ApplicationException پرتاب می کنند.

برنامه زیر طریقه ی استفاده از ReaderWriterLockSlim را نشان می دهد. سه نخ دائماً یک لیست را می شمارند ، مادامی که دو نخ بعدی در هر ثانیه یک عدد رندم را وارد لیست می کنند. یک قفل خواندن از خوانندگان لیست حمایت می کند و یک قفل نوشتن از نویسندگان لیست حفاظت می کند :

```
class SlimDemo
{
  static ReaderWriterLockSlim rw = new ReaderWriterLockSlim();
  static List<int> items = new List<int>();
  static Random rand = new Random();

  static void Main()
  {
    new Thread (Read).Start();
    new Thread (Read).Start();
    new Thread (Read).Start();

    new Thread (Write).Start ("A");
    new Thread (Write).Start ("B");
  }

  static void Read()
  {
    while (true)
    {
      rw.EnterReadLock();
      foreach (int i in items)
        Thread.Sleep (10);
      rw.ExitReadLock();
    }
  }

  static void Write (object threadID)
  {
    while (true)
    {
      int newNumber = GetRandNum (100);
      rw.EnterWriteLock();
```

```
      items.Add (newNumber);
      rw.ExitWriteLock();
      Console.WriteLine ("Thread " + threadID + " added " + newNumber);
      Thread.Sleep (100);
    }
  }

  static int GetRandNum (int max)
  {
    lock (rand)
      return rand.Next (max);
  }
}
```

در کد تولید شده ، شما معمولاً بلوک های try/finally را برای تضمین اینکه در صورت بروز یک خطا ، قفل ها آزاد شده اند اضافه می کنید.

نتیجه کد بالا در زیر آمده است :

```
Thread B added 61
Thread A added 83
Thread B added 55
Thread A added 33
...
```

کلاس ReaderWriterLockSlim اجازه ی فعالیت همزمان متد Read را نسبت به یک قفل ساده بیشتر فراهم می آورد. ما این کار را می توانیم با نوشتن کد زیر در متد Write ، در شروع حلقه ی while نشان دهیم.

```
Console.WriteLine (rw.CurrentReadCount + " concurrent readers");
```

این کد تقریباً همیشه عبارت "3 concurrent readers" را چاپ می کند (متد Read بیشتر زمان خود را درون حلقه ی foreach صرف می کند).

علاوه بر CurrentReadCount ، کلاس ReaderWriterLockSlim صفت های زیر را برای دیدن قفل ها فراهم می آورد :

- public bool IsReadLockHeld { get; }
- public bool IsUpgradeableReadLockHeld { get; }
- public bool IsWriteLockHeld { get; }
- public int WaitingReadCount { get; }
- public int WaitingUpgradeCount { get; }
- public int WaitingWriteCount { get; }
- public int RecursiveReadCount { get; }
- public int RecursiveUpgradeCount { get; }
- public int RecursiveWriteCount { get; }

بعضی موقع جابجایی یک قفل خواندن برای یک قفل نوشتن در یک عمل atomic تکی مفید می باشد. برای مثال ، فرض کنید شما می خواهید یک آیتم به یک لیست اضافه کنید به شرطی که این آیتم قبلاً در لیست وجود نداشته باشد. معمولاً شما مایلید تا زمان صرف شده برای نگه داشتن قفل نوشتن (انحصاری) را کم کنید ، بنابراین شما ممکن است چنین عمل کنید :

1- یک قفل خواندن ایجاد کنید
2- چک کنید که آیا آیتم در لیست وجود دارد ، و اگر چنین بود قفل را آزاد کرده و به برنامه بازگردید

3- قفل خواندن را آزاد کنید
4- یک قفل نوشتن ایجاد کنید
5- آیتم را به لیست اضافه کنید

مشکل این روش این است که بین گام های 3 و 4 نخ دیگری می تواند دزدکی وارد شده و لیست را تغییر دهد (مثلاً یک آیتم به لیست اضافه کند). کلاس ReaderWriterLockSlim این کار را با نوع سومی از قفل گذاری که قفل گذاری قابل توسعه (upgradeable lock) نام دارد ، انجام می دهد. یک قفل توسعه پذیر شبیه قفل خواندن می باشد با این تفاوت که این قفل بعداً می تواند به یک قفل نوشتن در یک عملیات atomic ، تبدیل شود. در زیر طریقه ی استفاده از آن بیان شده است :

1- EnterUpgareableReadLock را فراخوانی کنید
2- عملیات خواندن را انجام دهید (یعنی چک کنید که آیتم وجود دارد یا نه)
3- EnterWriteLock را فراخوانی کنید (با این کار قفل توسعه پذیر به قفل نوشتن تبدیل می شود)
4- عملیات نوشتن را انجام دهید (یعنی آیتم را به لیست اضافه کنید)
5- ExitWriteLock را فراخوانی کنید (با این کار قفل نوشتن را دوباره به قفل توسعه پذیر تبدیل می شود)
6- هر عملیات خواندن دیگری را که می خواهید انجام دهید
7- ExitUpgradealbeReadLock را فراخوانی کنید

از دید فراخواننده ، این کار شبیه یک قفل گذاری تودرتو یا بازگشتی است. خوشبختانه ، در گام شماره 3 ReaderWriterLockSlim به صورت atomic قفل خواندن شما را آزاد می کند و یک قفل نوشتن جدید فراهم می آورد.

یک تفاوت مهم دیگر بین قفل توسعه پذیر و قفل خواندن وجود دارد. مادامی که یک قفل توسعه پذیر می تواند با هر تعدادی از قفل های خواندن باشد ، فقط یک قفل توسعه پذیر می تواند در یک زمان بدست بیاید. این کار از تولید بن بست ها به وسیله تولید پیاپی تبدیلات شرکت کننده جلوگیری می کند – درست مانند بروزرسانی قفل ها که در SQL Server انجام می گیرد :

SQL Server	ReaderWriterLockSlim
اشتراک گذاری قفل	قفل خواندن
قفل انحصاری	قفل نوشتن
قفل بروزرسانی	قفل توسعه پذیر

ما می توانیم یک قفل توسعه پذیر را با تغییر متد Write در مثال قبلی به این صورت که فقط زمانی که آیتم مورد نظر در لیست نبود آنرا اضافه کند ، نشان دهیم :

```
while (true)
{
  int newNumber = GetRandNum (100);
  rw.EnterUpgradeableReadLock();
  if (!items.Contains (newNumber))
  {
    rw.EnterWriteLock();
    items.Add (newNumber);
    rw.ExitWriteLock();
    Console.WriteLine ("Thread " + threadID + " added " + newNumber);
  }
  rw.ExitUpgradeableReadLock();
```

```
  Thread.Sleep (100);
}
```

کلاس ReaderWriterLock همچنین تبدیل قفل را می تواند انجام دهد – اما به طور نامطمئن زیرا این کلاس از مفهوم قفل های توسعه پذیر حمایت نمی کند. به این دلیل است که طراح ReaderWriterLockSlim باید دوباره با یک کلاس جدید شروع به کار کند.

بازگشت قفل

معمولاً ، قفل گذاری تودرتو یا بازگشتی توسط ReaderWriterLockSlim منع شده است. از این رو ، کد زیر خطایی را تولید می کند :

```
var rw = new ReaderWriterLockSlim();
rw.EnterReadLock();
rw.EnterReadLock();
rw.ExitReadLock();
rw.ExitReadLock();
```

اگر شما ReaderWriterLockSlim را به صورت زیر بسازید ، این کد بدون خطا اجرا می شود :

```
var rw = new ReaderWriterLockSlim (LockRecursionPolicy.SupportsRecursion);
```

این کد تضمین می کند که اگر شما بخواهید قفل گذاری بازگشتی می تواند اتفاق بیفتد. قفل گذاری بازگشتی می تواند پیچیدگی نامطلوبی را تولید کند زیرا حاصل شدن بیشتر از یک نوع قفل ممکن است :

```
rw.EnterWriteLock();
rw.EnterReadLock();
Console.WriteLine (rw.IsReadLockHeld);     // True
Console.WriteLine (rw.IsWriteLockHeld);    // True
rw.ExitReadLock();
rw.ExitWriteLock();
```

شیوه اصلی این است که ، بار اولی که شما یک قفل را تولید کردید ، قفل های بازگشتی بعدی با توجه به معیار زیر می توانند کوچکتر شوند ، اما نمی توانند بزرگتر شوند :

قفل خواندن --> قفل توسعه پذیر --> قفل نوشتن

درخواستی برای توسعه یک قفل توسعه پذیر به یک قفل نوشتن ، در هر حال ، همیشه منطقی است.

Thread Pooling

اگر برنامه نخ های بسیاری دارد که بیشتر زمان خود را در انسدادی که توسط یک مدیر انتظار فراهم آمده است ، صرف می کنند ، شما می توانید سنگینی منابع را با استفاده از thread pooling کاهش دهید. یک مخزن نخ با یکی کردن بسیاری از مدیران انتظار به تعداد نخ های کمتر در منابع سیستم صرفه جویی می کند.

برای استفاده از مخزن نخ شما یک مدیر انتظار با یک delegate ثبت می کنید ، تا وقتی مدیر انتظار علامتی دریافت کرد اجرا شود. این کار با فراخوانی ThreadPool.RegisterWaitForSingleObject مانند مثال زیر ، انجام می گیرد.

```
class Test
{
  static ManualResetEvent starter = new ManualResetEvent (false);
```

```
  public static void Main()
  {
    ThreadPool.RegisterWaitForSingleObject (starter, Go, "hello", -1, true);
    Thread.Sleep (5000);
    Console.WriteLine ("Signaling worker...");
    starter.Set();
    Console.ReadLine();
  }

  public static void Go (object data, bool timedOut)
  {
    Console.WriteLine ("Started " + data);
    // Perform task...
  }
}
```

برنامه ی بالا خروجی زیر را تولید می کند :

```
(5 second delay)
Signaling worker...
Started hello
```

علاوه بر مدیر انتظار (Wait Handle) و delegate ، متد RegisterWaitForSingleObject یک شی "جعبه سیاه" نیز دریافت می کند که به متد delegate شما پاس داده می شود (تقریباً شبیه به یک ParameterizedThreadStart) ، همچنین یک زمان timeout بر حسب میلی ثانیه (-1 هیچ timeout ای را نشان نمی دهد) و یک پرچم که نشان می دهد که آیا درخواست بعد از دوباره ظاهر شدن انجام شده است یا نه.

همه ی نخ های مخزن شده نخ های پس زمینه هستند ، به این معنی که آنها به محض اینکه کار نخ های پیش زمینه به پایان برسد ، نابود می شوند. به هر حال اگر نخی بخواهد تا زمانی که هر عمل مهمی در نخ های مخزن قبل از وجود برنامه به پایان برسد ، منتظر بماند ، فراخوانی متد Join در نخ ها یک اختیار نخواند بود ، زیرا کار نخ های مخزن هرگز به پایان نمی رسد! ایده این است که آنها بازیابی شوند و فقط زمانی که پردازش مرکزی نابود شد ، به کار خود پایان دهند. بنابراین برای اینکه بدانیم چه موقع کار یک نخ درون مخزن به پایان رسیده است ، نخی باید علامت بدهد – برای نمونه به وسیله ی یک مدیر انتظار دیگر.

فراخوانی متد Abort برای نخ های مخزن ایده بدی است. نخ ها نیاز دارند تا در طول دامنه ی زندگی برنامه بازیابی شوند.

شما همچنین می توانید از مخزن نخ بدون یک مدیر انتظار و با فراخوانی متد QueueUserWorkItem استفاده کنید – یک delegate برای اجرای فوری تعیین کنید. با این کار شما نمی توانید نخ های به اشتراک گذاشته شده میان کارهای متعدد را نجات دهید ، اما سود بهتری نصیبتان می شود : مخزن نخ مقدار ماکزیممی را روی تعداد کل نخ های موجود در مخزن قرار داده است (25 ، به طور پیش فرض) ، اگر تعداد کارها بیشتر از این مقدار شود ، به طور اتوماتیک کارها به صف اضافه می شوند. این کار تقریباً شبیه یک صف تولید کننده / مصرف کننده (در بخش دو توضیح داده شده است) با 25 مصرف کننده می باشد! در مثال زیر ، 100 کار در یک مخزن نخ به صف شده اند ، 25 کار در هر زمان انجام می گیرد. نخ اصلی تا زمانی که آنها تماماً با استفاده از ساختار Wait and Pulse کامل شوند، منتظر می ماند :

```
class Test
{
  static object workerLocker = new object ();
  static int runningWorkers = 100;
```

```
  public static void Main()
  {
    for (int i = 0; i < runningWorkers; i++)
    {
      ThreadPool.QueueUserWorkItem (Go, i);
    }
    Console.WriteLine ("Waiting for threads to complete...");
    lock (workerLocker)
    {
      while (runningWorkers > 0)
        Monitor.Wait (workerLocker);
    }
    Console.WriteLine ("Complete!");
    Console.ReadLine();
  }

  public static void Go (object instance)
  {
    Console.WriteLine ("Started: " + instance);
    Thread.Sleep (1000);
    Console.WriteLine ("Ended: " + instance);
    lock (workerLocker)
    {
      runningWorkers--;
      Monitor.Pulse (workerLocker);
    }
  }
}
```

به منظور اینکه بیشتر از یک شی تنها به متد هدف پاس بدهیم ، یک نخ می تواند یک شی با همه ی صفت های مورد نیاز تعریف کند ، یا یک متد بی نام را فراخوانی کند. برای مثال ، اگر متد Go دو متغیر از نوع integer را قبول کند ، کد بالا می تواند با کد زیر شروع شود :

```
ThreadPool.QueueUserWorkItem (delegate (object notUsed) { Go (23,34); });
```

راه دیگری در مخزن نخ استفاده از Asynchronous delegate می باشد.

Asynchromous Delegates

در بخش 1 درباره ی طریقه ی پاس دادن اطلاعات به یک نخ با استفاده از ParameterizedThreadStart صحبت کردیم. بعضی موقع شما نیاز دارید تا به سمت دیگری بروید ، و وقتی اجرای آنها به پایان رسید مقادیر بازگشتی را از نخ ها دریافت کنید. Asynchronous Delegate ها یک مکانیزم ساده برای این کار ارائه می دهند ، به هر تعداد از آرگومان ها اجازه می دهند تا در هر دو مسیر پاس داده شوند. از این پس ، خطاهای مدیریت نشده در asynchronous delegate ها به راحتی دوباره به نخ اصلی پرتاب می شوند ، بنابراین به مدیریت مستقیم خطا ها نیازی نیست.
Asynchronous delegate ها همچنین راه دیگری برای مخازن نخ فراهم آورده اند.

بهایی که شما باید برای همه این موارد بپردازید ، در رابطه با مدل asynchronous می باشد. برای اینکه متوجه منظورم شوید ، ما ابتدا در مورد مدل برنامه نویسی معمول تری با نام synchronous صحبت می کنیم. ما می خواهیم دو صفحه ی وب را با هم مقایسه کنیم. ما این کار را می توانیم با دانلود هر صفحه انجام دهیم ، سپس خروجی ها را با هم چک کنیم :

```
static void ComparePages() {
  WebClient wc = new WebClient ();
  string s1 = wc.DownloadString ("http://www.oreilly.com");
  string s2 = wc.DownloadString ("http://oreilly.com");
  Console.WriteLine (s1 == s2 ? "Same" : "Different");
}
```

این کار می تواند سریعتر انجام شود به این صورت که هر دو صفحه در یک زمان دانلود شوند. راهی برای دیدن مشکل این است که از متد DownloadString برای مسدود کردن متد فراخواننده در زمان دانلود صفحه ایراد بگیریم. کار خوبی است که متد DownloadString را در یک مدل asysnchronous non-blocking فراخوانی کنیم. به عبارت دیگر :

1- ما به متد DownloadString می گوییم که اجرا شود
2- ما کار های دیگری را تا زمانی که این متد در حال کار کردن است انجام می دهیم ، مانند دانلود دیگر صفحات
3- ما از متد DownloadString برای نتیجه ی آن سؤال می پرسیم

کلاس WebClient در واقع متد DownloadStringAsync را فراخوانی می کند ، که عملی asynchronous گونه را فراهم می آورد. برای الان ، ما این متد را نادیده می گیریم و روی مکانیزم توانایی فراخوانی هر متد به صورت ناهماهنگ تمرکز می کنیم.

گام سوم ، آن چیزیست که asynchronous delegate ها را مفید کرده است. فراخواننده با کارگر برای گرفتن نتیجه و اجازه دادن به هر خطا برای دوباره پرتاب شدن ، قرار می گذارد. بدون این گام ، ما چند خطی عادی خودمان را داریم. مادامی که استفاده از asynchronous delegate ها بدون قرار ملاقاتی ممکن است ، شما کمتر به فراخوانی ThreadPool.QueueWorkerItem یا استفاده از BackgroundWorker نیاز پیدا می کنید.

در زیر مثالی از طریقه استفاده از asynchronous delegate برای دانلود دو صفحه وب آمده است ، در حالی که به طور همزمان یک عمل محاسبه را نیز انجام می دهد :

```
delegate string DownloadString (string uri);

static void ComparePages()
{

  // Instantiate delegates with DownloadString's signature:
  DownloadString download1 = new WebClient().DownloadString;
  DownloadString download2 = new WebClient().DownloadString;

  // Start the downloads:
  IAsyncResult cookie1 = download1.BeginInvoke (uri1, null, null);
  IAsyncResult cookie2 = download2.BeginInvoke (uri2, null, null);

  // Perform some random calculation:
  double seed = 1.23;
  for (int i = 0; i < 1000000; i++)
    seed = Math.Sqrt (seed + 1000);

 // Get the results of the downloads, waiting for completion if necessary.
 // Here's where any exceptions will be thrown:
 string s1 = download1.EndInvoke (cookie1);
```

```
  string s2 = download2.EndInvoke (cookie2);

  Console.WriteLine (s1 == s2 ? "Same" : "Different");
}
```

ما با اعلان و مقدار دهی به delegate هایی که می خواهیم تا به صورت asynchronous اجرا شوند شروع می کنیم. در این مثال ، ما به دو delegate به منظور اینکه هر کدام به یک شی WebClient جدا ارجاع داده شوند ، نیاز داشتیم (WebClient اجازه دسترسی همزمان را ندارد – اگر این اجازه را داشت ، ما می توانستیم از یک delegate در سرتاسر کد استفاده کنیم).

متد BeginInvoke به دو آرگومان نیازمند است – یک callback اختیاری و شی داده ؛ اگر نیازی به این آرگومان ها نبود آنها می توانند با مقدار null تنظیم شوند. متد BeginInvoke یک شی IAsynchResult را بازمی گرداند که مانند یک کوکی برای فراخوانی EndInvoke رفتار می کند. شی IAsynchResult دارای صفت IsCompleted می باشد که برای چک کردن پیشروی اجرا از آن استفاده می شود.

ما سپس متد EndInvoke را برای delegate فراخوانی می کنیم ، همان طور که نتایج آن مورد نیاز است. اگر لازم بود EndInvoke منتظر می ماند ، تا زمانی که کار متد پایان یابد ، سپس مقدار بازگشتی متد را همانطور که در delegate تعیین شده بود (در این مثال string) بازمی گرداند. یک ویژگی خوب EndInvoke این است که اگر متد DownloadString هر پارامتر ref یا out ای داشته باشد ، این آرگومان ها به امضای متد EndInvoke اضافه می شوند ، که با این کار اجازه برگشت مقادیر بیشتری از فراخواننده داده می شود.

اگر در هر نقطه از اجرای متد asynchronous خطایی رخ دهد ، به وسیله ی نخ فراخواننده و با فراخوانی EndInvoke دوباره پرتاب می شود. این کار مکانیزم منظمی برای هدایت خطا ها به فراخواننده را فراهم می آورد.

اگر متدی که شما به صورت asynchronous در حال فراخوانی آن می باشید هیچ مقدار بازگشتی ای نداشته باشد ، شما هنوز هم مجبور به اجرای متد EndInvoke می باشید. در یک دید کاربردی این قابلیت جای بحث دارد ؛ MSDN حرف های ضد و نقیضی از این قابلیت دارد. اگر شما می خواهید تا EndInvoke فراخوانی نشود ، شما نیاز دارید تا یک مدیریت خطا روی متد کارگر داشته باشید.

Asynchronous Methods

برخی انواع داده در .NET Framework. ورژن های asynchronous متدهایشان را عرضه می کنند ، با نام هایی که با Begin و End شروع می شود. این متدها ، متد های asynchronous نامیده می شوند و امضایی شبیه به asynchronous delegate ها دارد ، اما برای حل مشکلات سخت تری وجود دارند : برای اینکه فعالیت های همزمان کننده ی بیشتری را اجازه دهیم شما نخ ها را در اختیار دارید. یک وب یا سرور سوکت های TCP ، برای مثال ، چندین هزار درخواست همزمان را فقط از طریق تعدادی از نخ های وارد مخزن شده (pooled) می تواند پردازش کند ، البته اگر این نخ ها با استفاده از NetworkStream.BeginRead و NetworkStream.BeginWrite نوشته شوند.

مگر اینکه شما یک برنامه ی همزمانی بالا بنویسید ، در هر حال ، شما می بایست از متدهای asynchronous به دلایل زیر دوری کنید :

- بر خلاف asynchronous delegate ، متدهای asynchronous ممکن است به طور موازی با فراخواننده اجرا نشوند
- اگر شما نتوانید با دقت زیاد الگو را پی گیری کنید فایده ی متدهای asynchronous دیده نمی شود
- وقتی شما دقیقاً از الگو پیروی می کنید برخی مسائل بسیار سریع پیچیده می شوند

اگر شما به دنبال اجرای موازی هستید ، بهتر است تا ورژن synchronous متد را به وسیله ی یک asynchronous delegate فراخوانی کنیم (یعنی NetworkStream.Reak). انتخاب دیگر استفاده از BackgroundWorker یا ThreadPool.QueueUserWorkItem – یا حتی ساده تر تولید یک نخ جدید می باشد.

51 Asynchronous Events

الگوی دیگری که به وسیله ی انواع داده ها وجود دارد ، می تواند ورژن های Asynchronous متدهایمان را فراهم کند. این الگو "الگوی asynchronous بر پایه رویداد" نامیده می شود و با متدی که پایان نام آن با عبارت Async تمام می شود ، و رویدادی که نام آن با عبارت Completed به پایان می رسد ، تعیین می شود. کلاس WebClient از این الگو در متد DownloadStringAsync استفاده می کند. برای استفاده از این متد ، ابتدا رویداد Completed را می بایست مدیریت کنید (یعنی DownloadStringCompleted) و سپس متد Async را فراخوانی کنید (یعنی DownloadStringAsync). وقتی کار متد تمام شد ، این متد رویداد شما را فراخوانی می کند. بدبختانه ، پیاده سازی WebClient دارای نقص می باشد : متد هایی مانند DownloadStringAsync فراخواننده را برای بخشی از زمان دانلود مسدود می کند.

این الگو همچنین رویداد هایی را برای گزارش پیشروی و لغو اجرا در اختیار قرار می دهد ، که برای برنامه های ویندوزی ای که فرم ها و کنترل ها را بروزرسانی می کنند ، طراحی شده اند. اگر شما به این ویژگی ها در نوع داده ای که از این مدل حمایت نمی کند ، نیازمندید ، شما نمی بایست سنگینی پیاده سازی الگو را خودتان بر عهده بگیرید ، در هر حال (شما به این کار ها نیاز نخواهید داشت!) همه این اعمال بسیار ساده تر می تواند با کنترل BackgroundWorker انجام گیرد.

تایمر ها

راحت ترین راه برای اجرای یک متد به طور منظم استفاده از یک تایمر است – مانند کلاس Timer که در فضای نام System.Threading وجود دارد. تایمر نخ از مزیت مخزن نخ (thread pool) سود می برد. و به تایمر های متعدد بدون تولید overhead ای ناشی از زیاد بودن نخ ها ، اجازه ی تولید می دهد. تایمر کلاس بسیار ساده ای است ، دارای یک سازنده و فقط دو متد.

```
public sealed class Timer : MarshalByRefObject, IDisposable
{
  public Timer (TimerCallback tick, object state, 1st, subsequent);
  public bool Change (1st, subsequent);   // To change the interval
  public void Dispose();                  // To kill the timer
}
1st = time to the first tick in milliseconds or a TimeSpan
subsequent = subsequent intervals in milliseconds or a TimeSpan
 (use Timeout.Infinite for a one-off callback)
```

در مثال زیر ، یک تایمر متد Tick را هر 5 ثانیه فراخوانی می کند و عبارت "tick..." در صفحه نمایش چاپ می شود ، و تا زمانی که کاربر کلید Enter را فشار دهد ، برنامه به کار خود ادامه می دهد :

```
using System;
using System.Threading;

class Program
{
  static void Main()
  {
    Timer tmr = new Timer (Tick, "tick...", 5000, 1000);
    Console.ReadLine();
    tmr.Dispose();         // End the timer
  }
```

```
  static void Tick (object data)
  {
    // This runs on a pooled thread
    Console.WriteLine (data);        // Writes "tick..."
  }
}
```

NET Framework. کلاس تایمر دیگری را در همان فضای نام ایجاد کرده است. این کلاس قابلیت های بیشتر برای استفاده از مخزن نخ ، را نسبت به کلاس Timer فراهم می آورد. در زیر خلاصه ی ویژگی های اضافه شده به آن قرار دارد :

- پیاده سازی کلاس Component ، که به این کلاس اجازه می دهد تا در محیط طراحی ویژوال استودیو قرار گیرد
- دارای صفت Interval به جای استفاده از متد Change
- دارای یک رویداد Elapsed به جای استفاده از callback delegate
- دارای یک صفت به نام Enabled برای شروع یا توقف تایمر (مقدار پیش فرض آن false می باشد)
- دارای متد های Start و Stop ، در این موقعیت شما ممکن است توسط Enabled سردرگم شوید
- یک پرچم AutoReset برای نمایش یک رویداد دوباره ظاهر شونده (مقدار پیچ فرض آن true می باشد)

در زیر مثالی از استفاده از این کلاس قرار دارد :

```
using System;
using System.Timers;   // Timers namespace rather than Threading

class SystemTimer {
  static void Main() {
    Timer tmr = new Timer();       // Doesn't require any args
    tmr.Interval = 500;
    tmr.Elapsed += tmr_Elapsed;    // Uses an event instead of a delegate
    tmr.Start();                   // Start the timer
    Console.ReadLine();
    tmr.Stop();                    // Pause the timer
    Console.ReadLine();
    tmr.Start();                   // Resume the timer
    Console.ReadLine();
    tmr.Dispose();                 // Permanently stop the timer
  }

  static void tmr_Elapsed (object sender, EventArgs e) {
    Console.WriteLine ("Tick");
  }
}
```

NET Framework. همچنین کلاس دیگری از تایمر را در اختیار دارد – در فضای نام System.Forms. که در اینترفیس مورد استفاده شبیه کلاس System.Timers.Timer می باشد ، اما در کاری که انجام می دهد اساساً با آن فرق دارد. تایمر فرم های ویندوز از مخزن نخ استفاده نمی کند ، به جای آن همیشه رویداد Tick را از همان نخی که آنرا ایجاد کرده است ، فراخوانی می کند. فرض کنید این نخ ، نخ اصلی برنامه باشد –که مسئول تولید فرم ها و کنترل ها نیز می باشد – رویداد تایمر می تواند با فرم ها و کنترل ها بدون تخطی از امنیت نخ یا تحمیل apartment threading تعامل

کند. متد Control.Invoke نیاز نمی باشد. تایمر ویندوز در واقع یک تایمر تک نخه می باشد. همچنین یک تایمر تک نخه نیز برای WPF وجود دارد ، که DispatcherTimer نامیده می شود.

تایمر های فرم های ویندوز و WPF برای کارهایی شامل بروزرسانی واسط کاربری و اجرای سریع استفاده می شوند. اجرای سریع عمل مهمی است زیرا رویداد Tick در نخ اصلی برنامه رخ می دهد – که اگر صورت نپذیرد باعث می شود تا واسط کابری غیر قابل پاسخ شود.

مخزن محلی

یک نخ ، یک مکان ذخیره اطلاعات جدا از دیگر نخ ها می گیرد. این مکان برای ذخیره اطلاعات خارج از محدوده ی کاری نخ مفید می باشد – که از زیر ساخت های مسیر اجرایی مانند پیام رسانی ، تراکنش یا token های محرمانه ، حمایت می کند. پاس دادن چنین اطلاعاتی به عنوان آرگومان ورودی یک تابع کار خطرناکی است زیرا مسیر اجرای خود متد را نیز منحرف خواهد کرد ؛ ذخیره این اطلاعات در متغیرهای استاتیک به معنی اشتراک آن برای همه نخ ها می باشد.

متد Thread.GetData از اطلاعات را مکان ذخیره ی اطلاعات نخ می خواند ؛ متد Thread.SetData اطلاعات را در این مکان می نویسد. هر دو متد برای شناسایی یک شکاف برای ورود به یک شی از کلاس LocalDataStoreSlot نیازمندند – این شکاف (slot) در واقع یک رشته ی متنی می باشد که به این نام نامگذاری شده است – همین خطی مشی برای همه نخ ها می تواند استفاده شود و این نخ ها همچنان مقادیر مربوط به خود را می توانند دریافت کنند. در زیر مثالی نشان داده شده است :

```
class ... {
  // The same LocalDataStoreSlot object can be used across all threads.
  LocalDataStoreSlot secSlot = Thread.GetNamedDataSlot ("securityLevel");

  // This property has a separate value on each thread.
  int SecurityLevel {
    get {
      object data = Thread.GetData (secSlot);
      return data == null ? 0 : (int) data;    // null == uninitialized
    }
    set {
      Thread.SetData (secSlot, value);
    }
  }
  ...
```

متد Thread.FreeNamedDataSlot یک شکاف (slot) اطلاعاتی را برای همه نخ ها آزاد می کند – اما فقط برای یک بار همه ی اشیای LocalDataStoreSlot هم نام از بین می روند. این کار به نخ ها تضمین می دهد که شکاف های اطلاعاتی را از بند خود خارج نکنند – به شرطی که آنها ارجاعی به شی LocalDataStoreSlot داشته باشند و تا زمانی که از آن استفاده می کنند ، آنرا حفظ کنند.

فصل چهارم : مباحث پیشرفته

Non-Blocking Synchronization

قبلاً در بخش دو گفتیم که نیاز برای عمل همزمانی (synchronization) حتی در موقعیت های ساده ای نظیر عمل انتساب یا اضافه کردن یک فیلد ، احساس می شود. اگرچه عمل قفل گذاری می تواند این نیاز را برآورده کند ، اما قرار دادن یک قفل به این معنی است که یک نخ باید مسدود شود ، که باعث overhead و تاخیر در دوباره زمانبندی شدن آن نخ می شود. ساختارهای

Non-Blocking Synchronization ای که .NET Framework. فراهم آورده است ، می تواند اعمال ساده ای را بدون هیچ انسداد ، توقف ، یا انتظاری برای نخ انجام دهید. این ساختارها شامل استفاده از سازه هایی هستند که شدیداً ریز (atomic) هستند ، و شامل شکل دادن به کامپایلر برای استفاده از گرامر خواندنی/نوشتنی volatile می باشد. بعضی موقع این ساختارها در طریقه ی استفاده می توانند ساده تر از یک قفل باشند.

Atomicity و Interlocked

یک دستور زمانی atomic خوانده می شود که به صورت یک ساختار غیر قابل تقسیم اجرا شود. Atomicity شدید از هر تقابل و پیشدستی ای در نخ ها جلوگیری می کند. در C# ، عمل انتساب به یا خواندن از یک فیلد 32 بیتی یا کمتر(CPU 32 بیتی فرض شده است) یک عمل atomic خوانده می شود.

عملیات در فیلدهای بزرگتر atomic خوانده نمی شود ، که شامل دستوراتی است که از ترکیب چند عمل خواندن/نوشتن تشکیل شده باشد.

```
class Atomicity
{
  static int x, y;
  static long z;

  static void Test()
  {
    long myLocal;
    x = 3;                 // Atomic
    z = 3;                 // Non-atomic (z is 64 bits)
    myLocal = z;           // Non-atomic (z is 64 bits)
    y += x;                // Non-atomic (read AND write operation)
    x++;                   // Non-atomic (read AND write operation)
  }
}
```

عمل خواندن و نوشتن فیلد های 64 بیتی در CPU های 32 بیتی یک عمل atomic به حساب نمی آید زیرا این فیلد ها از دو مکان حافظه ای 32 بیتی استفاده می کنند. اگر نخ A یک مقدار 64 بیتی را بخواند در حالی که نخ B در حال بروزکردن آن فیلد می باشد ، کار نخ A ممکن است با ترکیب بیتی ای مقدار قدیمی و مقدار جدید فیلد پایان یابد.

اپراتورهای یگانه مانند x++ نیاز دارند تا ابتدا عمل خواندن را انجام دهند ، سپس آنرا پردازش کنند ، سپس دوباره اطلاعات را در متغیر بنویسند.

کلاس زیر را در نظر بگیرید :

```
class ThreadUnsafe
{
  static int x = 1000;
  static void Go ()
```

```
  {
    for (int i = 0; i < 100; i++)
      x--;
  }
}
```

اگر 10 نخ به طور همزمان متد Go را اجرا کنند ، شما انتظار دارید که مقدار x برابر صفر شود. اما برای این نتیجه تضمینی وجود ندارد ، زیرا نخی ممکن است میان اعمال خواندن مقدار جاری x ، کاهش آن و نوشتن دوباره در فیلد x ، از نخ دیگر پیشدستی کند (نتیجه این است که یک مقدار نامعلوم در فیلد نوشته خواهد شد).

یک راه برای حل این مشکلات این است که اعمال غیر atomic را در یک بلوک lock انجام دهیم. قفل گذاری در حقیقت ، atomicity را شبیه سازی می کند. کلاس Interlocked ، یک راه حل ساده تر و سریعتر برای اعمال atomic ساده فراهم آورده است :

```
class Program
{
  static long sum;

  static void Main()
  {                                   // sum

    // Simple increment/decrement operations:
    Interlocked.Increment (ref sum);                  // 1
    Interlocked.Decrement (ref sum);                  // 0

    // Add/subtract a value:
    Interlocked.Add (ref sum, 3);                     // 3

    // Read a 64-bit field:
    Console.WriteLine (Interlocked.Read (ref sum));           // 3

    // Write a 64-bit field while reading previous value:
    // (This prints "3" while updating sum to 10)
    Console.WriteLine (Interlocked.Exchange (ref sum, 10));     // 10

    // Update a field only if it matches a certain value (10):
    Interlocked.CompareExchange (ref sum, 123, 10);           // 123
  }
}
```

به طور کلی استفاده از کلاس Interlock بسیار مؤثرتر از استفاده از دستور lock است ، زیرا این کلاس هیچ وقت انسداد را تولید نمی کند و مشکلات overhead نخ هایی که موقتاً متوقف شده اند ، را ندارد.

کلاس Interlock همچنین برای پردازش های متعدد طولی نیز مناسب می باشند – در مقابل دستور lock ، فقط برای نخ هایی که در پردازش حال حاظر قرار دارند مناسب می باشد. مثالی از کاربرد این کلاس در عملیات خواندن و نوشتن مقادیر در حافظه های به اشتراک گذاشته شده می باشد.

موانع حافظه (Memory Barrier) و Volatility

کلاس زیر را در نظر بگیرید :

```
class Unsafe
 {
  static bool endIsNigh, repented;

  static void Main()
{
    new Thread (Wait).Start();       // Start up the spinning waiter
    Thread.Sleep (1000);             // Give it a second to warm up!
    repented = true;
    endIsNigh = true;
    Console.WriteLine ("Going...");
  }

  static void Wait()
{
    while (!endIsNigh);              // Spin until endIsNigh
     Console.WriteLine ("Gone, " + repented);
  }
}
```

اینجا یک سؤال مطرح می شود : یک وقفه مهم می تواند "Going" را از "Gone" جدا کند – به عبارت دیگر ، آیا برای متد Wait این امکان وجود دارد تا عمل spin را در حلقه ی while بعد از true شدن مقدار endIsNigh ، ادامه دهد ؟ بنابراین ، آیا برای متد Wait این امکان وجود دارد تا عبارت "Gone,false" را چاپ کند ؟

از لحاظ علمی اگر زمانبند نخ دو نخ را در CPU های مختلف مقداردهی کند (caching) ، جواب هر دو سؤال در ماشین های چند پردازنده بله است. فیلدهای repented و endIsNigh می توانند در رجیسترهای CPU ، برای بالابردن کارایی قرار بگیرند ، البته با یک وقفه ذاتی قبل از به روز رسانی مقادیر هنگام نوشتن در حافظه. وقتی رجیسترهای CPU دوباره در حافظه نوشته می شوند ، ترتیبی که آنها اساساً به روز می شدند ، زیاد مهم نیست.

عمل caching می تواند با استفاده از متدهای استاتیک Thread.VolatileRead و Thread.VolatileWrite برای خواندن و نوشتن در فیلدها ، انجام گیرد. VolatileRead به معنی خواندن آخرین مقدار است ؛ VolatileWrite به معنای نوشتن فوری در حافظه است. همین اعمال می تواند با ظرافت بیشتری با استفاده از کلمه ی کلیدی volatile انجام گیرد :

```
class ThreadSafe {
  // Always use volatile read/write semantics:
  volatile static bool endIsNigh, repented;
  ...
```

استفاده از کلمه ی کلیدی volatile دارای اولویت بیشتری نسبت به استفاده از توابع VolatileRead و VolatileWrite می باشد، اما می توان آنرا به صورت ساده تری نیز بیان کرد ، " این فیلد را thread-cache در نظر نگیرد".

همین نتایج را می توان با قرار دادن متغیرهای repented و endIsNigh در یک بلوک lock بدست آورد. این مطالب به خاطر یک اثر قفل گذاری است که یک مانع حافظه ایجاد می کند – ضمانت که volatility یک فیلد در بلوک lock از آن استفاده می کند ، در خارج حوزه ی این بلوک بسط داده نمی شود. به عبارت دیگر ، فیلدها در ورود به قفل (volatile read) تازه می شوند و قبل از خروج از قفل در حافظه نوشته می شوند (volatile write).

اگر ما بخواهیم به فیلدهای end و endIsNigh به صورت atomic دسترسی داشته باشیم ، استفاده از دستور lock در حقیقت لازم می باشد. مثلاً ، برای اجرای کدی شبیه زیر :

```
lock (locker)
{
  if (endIsNigh)
    repented = true;
}
```

وقتی فیلدی دفعات بسیاری در یک حلقه استفاده می شود ، استفاده از دستور lock ممکن است ترجیح داده شود (فرض کنید قفل تا پایان حلقه نگه داشته می شود). مادامی که یک عمل volatile read/write به یک قفل در کارایی غالب می شود ، بعید است که هزار عمل volatile read/write به یک قفل غالب شود!

Volatility فقط برای انواع داده ای اصلی و اشاره گرهای ناامن (unsafe pointer) مناسب می باشد – دیگر انواع داده در رجیسترهای CPU اصطلاحاً cache نمی شوند و نمی توانند با کلمه ی کلیدی volatile اعلان شوند. گرامر خواندن و نوشتن volatile هنگامی که به فیلدها با استفاده از کلاس Interlocked دست می یابیم ، به طور اتوماتیک بکار می رود.

اگر کسی عادت به استفاده از دستور lock برای دستیابی به فیلدها به وسیله ی نخ های متعدد دارد ، بنابراین استفاده از گرامر volatile و Interlocked لازم نمی باشد.

متدهای Wait و Pulse

در فصل 2 ما در مورد Wait Handle ها صحبت کرده بودیم – یک مکانیزم علامت دهی ساده که باعث انسداد یک نخ تا دریافت یک اخطار از نخ دیگر می شد.

ساختار علامت دهی قدرتمندتری توسط کلاس Monitor فراهم آمده است – به وسیله ی دو متد استاتیک Wait و Pulse. قاعده ی کلی این است که شما منطق علامت دهی را با استفاده از پرچم ها و فیلدهای دستی می نویسید (در مقایسه با دستوات قفل گذاری) ، سپس از متدهای Wait و Pulse برای تسهیل عمل spin مربوط به CPU استفاده می کنیم. فایده ی استفاده از این خطی مشی سطح پایین این است که شما فقط با استفاده از دستورات Wait و Pulse و lock می توانید به کارایی AutoResetEvent ، ManualResetEvent و Semaphore و نیز متدهای استاتیک WaitAll و WaitAny مربوط به WaitHandle دست یابید. بنابراین متد های Wait و Pulse می توانند جوابگوی موقعیت هایی که دیگر Wait Handler ها به سختی انجام می دهند ، باشند.

مشکلی که متدهای Wait و Pulse دارند ، مستندات ضعیف آنها می باشد و اینکه اوضاع را بدتر می کنند ، اگر شما آنها را بدون یک فهم کامل فراخوانی کنید ، آنها متوجه می شوند – و از گرداندن شما در خارج لذت می برند و شما را اذیت می کنند! خوشبختانه ، یک الگوی ساده وجود دارد که می توان از آن در یک برنامه ی ناامن (fail-safe) در هر موقعیتی استفاده کرد.

Wait و Pulse تعریف شده

هدف متدهای Wait و Pulse فراهم آوردن یک مکانیزم علامت دهی ساده است : متد Wait نخ را تا دریافت اخطار از دیگر نخ ها به حالت انتظار در می آورد ؛ متد Pulse این اخطار را ایجاد می کند.

برای درست اجرا شدن عمل علامت دهی متد Wait باید قبل از متد Pulse فراخوانی شود. اگر اول متد Pulse فراخوانی شود ، پالس مربوط به آن نخ از بین می رود ، و waiter اخیر باید منتظر یک پالس جدید باشد ، و یا برای همیشه مسدود شده باقی بماند. این کار با رفتار AutoResetEvent تفاوت دارد ، که در آن متد Set اثر محکم کاری داشت و بنابراین اگر قبل از متد WaitOne فراخوانی می شد دارای اثر مفیدی بود.

ممکن است کسی از یک شی همزمان کننده به هنگام فراخوانی Wait یا Pulse استفاده کند. اگر دو نخ از همان شی استفاده کنند ، آنها قادرند تا به یکدیگر علامت بدهند. شی همزمان کننده باید در قبل از فراخوانی متدهای Wait یا Pulse قفل شود.

برای مثال اگر x دارای این اعلان باشد :

```
class Test
{
  // Any reference-type object will work as a synchronizing object
  object x = new object();
}
```

سپس کد زیر دستور ورودی مربوط به Monitor.Wait را مسدود می کند :

```
lock (x)
   Monitor.Wait (x);
```

کد زیر (اگر بعداً در نخ دیگری اجرا شود) نخ مسدود شده را آزاد می کند :

```
lock (x)
   Monitor.Pulse (x);
```

تغییر وضع دادن قفل (lock toggling)

برای این کار ، Monitor.Wait موقتاً ایجاد شده است یا تا زمانی که نخ در حالت انتظار است ، قفل را اصطلاحاً toggle می کند، بنابراین دیگر نخ ها (مانند نخی که عمل Pulse را انجام می دهد) می تواند از آن استفاده کند. متد Wait را می توان با شبه کدهای زیر شبیه سازی کرد :

```
Monitor.Exit (x);           // Release the lock
wait for a pulse on x
Monitor.Enter (x);          // Regain the lock
```

از اینرو یک Wait می تواند دو بار مسدود شود : یک بار هنگام انتظار برای یک پالس و دیگری برای غالب شدن بر قفل انحصاری. این همچنین به این معنی است که Pulse کاملاً یک نخ منتظر را از حالت انسداد خارج نمی کند : فقط وقتی عمل پالس یک نخ نخ زا از دستور lock خارج کند ، نخ منتظر می تواند واقعاً به کار خود ادامه دهد.

toggle کننده ی قفل Wait صرفنظر از استفاده از قفل های تودرتو ، مؤثر است. اگر Wait درون دو دستور lock تودرتو فراخوانی شود :

```
lock (x)
  lock (x)
    Monitor.Wait (x);
```

سپس Wait منطقاً به کدهای زیر باز می شود :

```
Monitor.Exit (x);
Monitor.Exit (x);    // Exit twice to release the lock
wait for a pulse on x
```

```
Monitor.Enter (x);
Monitor.Enter (x);  // Restore previous nesting level
```

که با گرامر رایج قفل گذاری سازگار است ، فقط اولین فراخوانی برای Monitor.Enter شانسی برای یک انسداد حاصل می کند.

چرا قفل؟

چرا متد های Wait و Pulse طراحی شده اند در حالی که آنها فقط در یک قفل کاربرد دارند ؟ دلیل اصلی این است که متد Wait می تواند بر حسب موقعیت فراخوانی شود – بدون به خطر افتادن امنیت نخ. برای اینکه مثال ساده ای را ببینیم ، فرض کنید ما می خواهیم فقط زمانی که یک فیلد بولین که available نامیده می شود ، مقدار false را دریافت کرد ، متد Wait را فراخوانی کنیم. کد زیر یک کد thread-safe است :

```
lock (x)
{
  if (!available)
    Monitor.Wait (x);
  available = false;
}
```

چندین نخ می توانند کد بالا را به طور همزمان اجرا کنند و هیچ یک در حین چک کردن مقدار فیلد available و فراخوانی متد Monitor.Wait از دیگری پیشدستی نمی کند. دو دستور به طور مؤثری atomic هستند. اخطاردهنده ای شبیه به بالا نیز thread-safe خواهد بود :

```
lock (x)
  if (!available)
 {
    available = true;
    Monitor.Pulse (x);
  }
```

تعیین یک زمان پایان (timeout)

می توان هنگام فراخوانی متد Wait یک زمان خروج برای آن نیز تعیین کرد ، هم به صورت میلی ثانیه و هم با استفاده از کلاس TimeSpan. در این صورت متد Wait مقدار false را زمانی که این زمان به پایان برسد ، بازمی گرداند. Timeout فقط برای عبارت "*انتظار*" (انتظار برای یک پالس) به کار می رود: متعاقباً متد Wait ای که با یک زمان پایان همراه است به منظور حصول دوباره ی قفل مسدود خواهد شد ، اهمیتی ندارد که چه قدر طول می کشد تا این کار را انجام دهد.

در اینجا مثالی آمده است :

```
lock (x) {
  if (!Monitor.Wait (x, TimeSpan.FromSeconds (10)))
    Console.WriteLine ("Couldn't wait!");
  Console.WriteLine ("But hey, I still have the lock on x!");
}
```

این منطق برای این رفتار همانی است که در یک برنامه ی Wait/Pulse با طراحی مناسب به کار می رود ، شی ای که متدهای Wait و Pulse را فراخوانی می کند ، به طور خلاصه مسدود است. بنابراین حصول دوباره ی قفل باید عملی لحظه ای و فوری باشد.

پالس دادن و تصدیق (acknowledgement)

ویژگی مهم Monitor.Pulse این است که این متد به صورت asynchronous اجرا می شود ، یعنی خود این متد به هیچ وجه مسدود یا متوقف نمی شود. اگر نخ دیگری در حال انتظار برای شی پالس شده باشد ، اخطار داده می شود ، وگرنه پالس هیچ اثری نخواهد داشت و بدون هیچ اخطاری نادیده گرفته می شود.

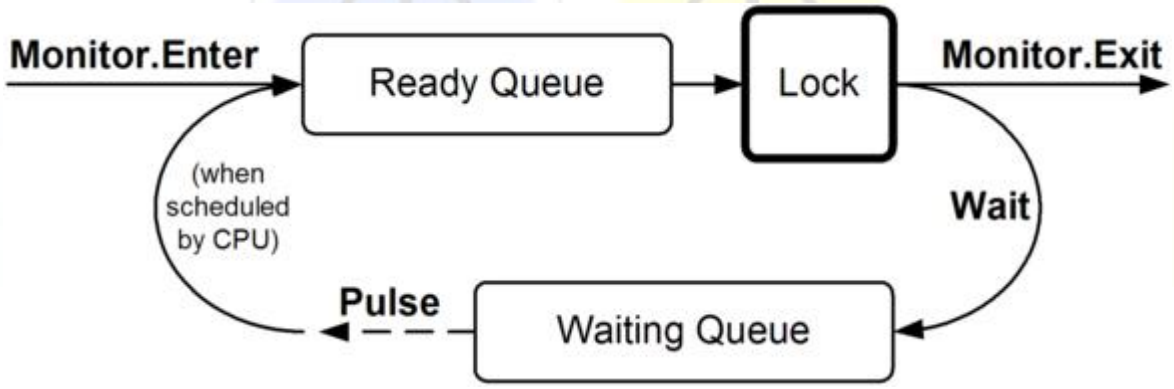
متد Pulse یک ارتباط یک طرفه را فراهم می کند : یک نخ پالس کننده به یک نخ منتظر را علامت می دهد. هیچ مکانیزم تصدیق ذاتی ای وجود ندارد : متد Pulse مقداری را که نشان دهنده ی این باشد که آیا پالس نخ دریافت شده است یا نه ، بازنمی گرداند. بنابراین، وقتی یک اخطار دهنده به قفل خود پالس می دهد و آنرا آزاد می کند ، از نظر زمانبند نخ ، هیچ ضمانتی روی یک نخ منتظر قابل انتخاب که به برنامه فوراً وارد شود ، وجود ندارد. یک وقفه ی مطلق می تواند وجود داشته باشد—در زمانی که هیچ نخی قفلی نداشت. این کار فهمیدن زمانی که یک نخ منتظر دوباره شروع به کار می کند ، را مشکل می کند ، مگر اینکه نخ منتظر به طور ویژه ای تصدیق شود ، برای نمونه به وسیله ی یک پرچم.

اگر یک تصدیق معتبر نیاز باشد ، این کار باید توسط ورود مستقیم کد انجام گیرد ، معمولاً به وسیله ی یک پرچم در مقایسه با متدهای Wait و Pulse .

با استناد به فعالیت زمان بر یک نخ منتظر که هیچ مکانیزم تصدیقی ندارد این نخ به عنوان "از دست رفته" به وسیله ی متدهای Wait و Pulse حساب می شود. شما باخته اید!

صف های انتظار و PulseAll

بیشتر از یک نخ می تواند روی یک شی به طور همزمان متد Wait را فراخوانی کند – در این موقعیت یک "صف نخ" در پشت شی همزمان کننده قرار می گیرد (این موضوع با "صف شروع کار(ready queue)" که برای اعطای دستیابی به یک قفل مورد استفاده قرار می گرفت ، تضاد دارد). هر Pulse یک نخ ار از ابتدای صف انتظار آزاد می کند ، بنابراین این نخ می تواند به صف شروع کار وارد شده و دوباره قفل را حاصل کند. به این موضوع به عنوان یک پارکینگ اتومبیل اتوماتیک فکر کنید : شما ابتدا در صف پرداخت برای معتبر کردن بلیط خود قرار می گیرید (صف انتظار) ؛ شما سپس در جلوی دروازه مانع که به شما اجازه ی خروج می دهد ، صفی را تشکیل می دهد (صف شروع کار).



شکل 2-صف انتظار و صف شروع به کار

ترتیبی که در ذات ساختار صف قرار دارد ، به هر حال ، اغلب در برنامه های Wait/Pulse مهم در نظر گرفته نمی شود ، و در این موقعیت ها بهتر است که مخزنی از نخ های منتظر را فرض کنیم. هر پالس ، یک نخ منتظر را از مخزن آزاد می کند.

کلاس Monitor همچنین یک متد با نام PulseAll نیز فراهم آورده است که صف یا مخزن درونی نخ ها منتظر را با حرکت سریعی آزاد می کند. نخ های پالس شده در همان زمان که پالس شدند ، شروع به اجرا نخواند کرد ، اما در عوض بعد از طی یک دنباله ی مرتب می تواند اجرا شود ، همانطور که هیچ دستور Wait ای تلاشی برای دوباره بدست آوردن همان قفل نمی کند. در حقیقت ، متد PulseAll نخ ها را از صف انتظار به صف شروع کار حرکت می دهد ، بنابراین آنها می توانند در یک الگوی ترتیبی دوباره شروع به کار کنند.

چه طور از متدهای Wait و Pulse استفاده کنیم

در زیر چگونگی شروع به کار ما آمده است. دو قانون برای این کار وجود دارد :

- اولین ساختار همزمان کننده که در دسترس ما قرار دارد دستور lock ، Monitor.Enter و Monitor.Exit می باشد
- هیچ محدودیتی روی spin در CPU وجود ندارد!

با این قوانین ، مثال زیر را در نظر بگیرید : یک نخ کارگر که تا زمان دریافت یک اخطار از نخ اصلی به حالت انتظار در آمده است:

```
class SimpleWaitPulse
 {
  bool go;
  object locker = new object();

  void Work()
  {
    Console.Write ("Waiting... ");
    lock (locker)
    {                    // Let's spin!
      while (!go)
      {
        // Release the lock so other threads can change the go flag
        Monitor.Exit (locker);
        // Regain the lock so we can re-test go in the while loop
        Monitor.Enter (locker);
      }
    }
    Console.WriteLine ("Notified!");
  }

  void Notify() // called from another thread
  {
    lock (locker)
    {
      Console.Write ("Notifying... ");
      go = true;
    }
  }
}
```

در زیر متد Main برای تنظیم مقادیر وجود دارد :

```
static void Main()
```

```
{
  SimpleWaitPulse test = new SimpleWaitPulse();

  // Run the Work method on its own thread
  new Thread (test.Work).Start();          // "Waiting..."

  // Pause for a second, then notify the worker via our main thread:
  Thread.Sleep (1000);
  test.Notify();               // "Notifying... Notified!"
}
```

متد Work جایی است که ما عمل spin را انجام می دهیم – به طور نامعقولی زمان CPU با تولید حلقه تا زمانی که متد go برابر true باشد ، مصرف می شود! در این حلقه ما می بایست toggle شدن قفل را حفظ کنیم – با متد های Exit و Enter مربوط به کلاس Monitor می توانیم آنها را دوباره حاصل کنیم یا آزاد کنیم – چنانچه دیگر نخ ها در حال اجرای متد Notify هستند ، خودشان می توانند قفل را بگیرند و پرچم go را ویرایش کنند. فیلد go به اشتراک گذاشته شده باید همیشه از درون یک قفل برای اجتناب از مشکلات volatility قابل دستیابی باشد (به یاد بیاورید که همه ی ساختارهای همزمان کننده ی دیگر ، مانند کلمه ی کلیدی volatile ، در این مرحله از طراحی خارج از گردانه رقابت می باشند!)

گام بعدی اجرای برنامه ی بالا امتحان اینکه این برنامه واقعاً کار می کند. در زیر خروجی حاصل از متد Main قرار داده شده است:

```
Waiting... (pause) Notifying... Notified!
```

حالا ما می توانیم متدهای Wait و Pulse را معرفی کنیم. ما این کار را به وسیله ی :

- جایگزین کردن lock toggling (Monitor.Exit که بعد از Monitor.Enter قرار گرفته است) با Monitor.Wait
- وارد کردن یک فراخوانی برای Monitor.Pulse وقتی که شرط انسداد تغییر می کند (یعنی زمانی که پرچم go ویرایش می شود).

در زیر کلاس بروز شده، به همراه حذف دستورات Console ای برای مختصر کردن کد ، قرار دارد :

```
class SimpleWaitPulse
{
  bool go;
  object locker = new object();

  void Work()
  {
    lock (locker)
      while (!go) Monitor.Wait (locker);
  }

  void Notify()
  {
    lock (locker)
    {
      go = true;
      Monitor.Pulse (locker);
    }
  }
}
```

کلاس بالا همانطور که قبلاً کار می کرد ، رفتار می کند ، اما با حذف spinning، دستور Wait به طور غیر مستقیم کار کدی که ما حذف کردیم را انجام می دهد – Monitor.Exit که بعد از Monitor.Enter قرار گرفته بود ، اما با گامی اضافه در میانه : زمانی که قفل آزاد می شود ، این قفل منتظر نخ دیگری برای فراخوانی متد Pulse باقی می ماند. متد اخطار دهنده همین کار را بعد از تنظیم شدن متغیر go با مقدار true انجام می دهد. کار انجام شد.

تعمیم دادن متدهای Wait و Pulse

حالا با هم این الگو را گسترش می دهیم. در مثال قبلی ، موقعیت انسداد ما شامل یک فیلد بولین بود – پرچم go، ما می توانیم ، در یک سناریوی دیگر ، یک پرچم اضافی دیگر را در نظر بگیریم که به وسیله ی نخ های منتظر زمانی که این نخ آماده ی شروع به کار یا کامل شدن است ، تنظیم شود. اگر ما بر این اساس کار خود را انجام دهیم که هر تعدادی از فیلدها در هر تعداد شرط انسداد می توانند قرار گیرند ، برنامه می تواند به شبه کد زیر تعمیم داده شود (فرم دارای spin) :

```
class X {
  Blocking Fields:  one or more objects involved in blocking condition(s), eg
   bool go;   bool ready;   int semaphoreCount;   Queue <Task> consumerQ...

  object locker = new object();     // protects all the above fields!

  ... SomeMethod {
    ... whenever I want to BLOCK based on the blocking fields:
    lock (locker)
      while (! blocking fields to my liking ) {
        // Give other threads a chance to change blocking fields!
        Monitor.Exit (locker);
        Monitor.Enter (locker);
      }

    ... whenever I want to ALTER one or more of the blocking fields:
    lock (locker) { alter blocking field(s) }
  }
}
```

ما سپس متدهای Wait و Pulse را همانطور که قبلاً اعمال کردیم ، در اینجا قرار می دهیم :

- در حلقه های انتظار ، lock toggling با Monitor.Wait تعویض می شود
- هروقت یک شرط انسداد تغییر کرد ، Pulse قبل از آزاد شدن قفل فراخوانی می شود.

در زیر شبه کد بروز شده قرار دارد :

متن استاندارد Wait/Pulse #1 : استفاده ی پایه ای از Wait/Pulse

```
class X {
  < Blocking Fields ... >
  object locker = new object();

  ... SomeMethod {
    ...
    ... whenever I want to BLOCK based on the blocking fields:
```

```
    lock (locker)
      while (! blocking fields to my liking )
        Monitor.Wait (locker);

    ... whenever I want to ALTER one or more of the blocking fields:
    lock (locker) {
      alter blocking field(s)
      Monitor.Pulse (locker);
    }
  }
}
```

این کار یک الگوی قدرتمند برای استفاده از متدهای Wait و Pulse ایجاد می کند. در زیر ویژگی های کلیدی این الگو قرار دارد :

- شرط های انسداد با استفاده از فیلدهای دستی پیاده سازی می شوند (که با عملکردی بدون استفاده از Wait و Pulse سازگار است ، اگرچه با spinning نیز این گونه است)
- متد Wait همیشه درون حلقه ی while فراخوانی می شود که شرط انسداد آنرا چک می کند (خود این متد درون یک دستور lock)
- یک شی همزمان کننده (در مثال بالا ، locker) برای همه ی Wait ها و Pulse ها ، و برای حفاظت از دستیابی به همه ی اشیا که در همه ی شرط های انسداد دخیل بودند ، استفاده می شود
- قفل ها به طور خلاصه مورد استفاده نگه داشته می شوند

از همه مهم تر ، با این الگو ، عمل پالسینگ یک نخ منتظر را مجبور به ادامه کار نمی کند. در عوض ، این الگو به نخ اخطار می دهد که بعضی تغییرات ایجاد شده است و به نخ توصیه می کند تا شرط انسداد خود را دوباره چک کند. نخ منتظر سپس تصمیم می گیرد که آیا او باید به کار خود ادامه دهد یا نه (به وسیله ی پیمایش دیگری روی حلقه ی while). فایده این کار این است که اجازه ی استفاده از شرط های پیچیده ی انسداد را بدون پیچیده کردن منطق همزمان کننده ، به ما می دهد.

دیگر فایده ی این الگو مصونیت نسبت به اثرات پالس های اشتباه است. یک پالس اشتباهی زمانی اتفاق می افتد که متد Pulse قبل از متد Wait فراخوانی شود – شاید به رقابتی که بین نخ اخطار دهنده و نخ منتظر وجود دارد وابسته است. اما چون در این الگو یک پالس به معنی "دوباره شرط انسداد خود را چک کن" (و اینکه "به کار خود ادامه نده") می باشد ، یک پالس زودهنگام به طور کاملاً امنی می تواند نادیده گرفته شود زیرا شرط انسداد قبل از فراخوانی متد Wait همواره در حال چک شدن است ، از حلقه while از این بابت تشکر می کنیم!

با این طراحی ، کسی می تواند فیلدهای انسداد متعددی تعریف کند ، و به آنها اجازه ی شرکت در شرط های انسداد متعدد را بدهد، و هنوز هم از یک شی همزمان کننده در سرتاسر کد (در مثال بالا شی locker) استفاده کند. این خطی مشی بهتر از این است که ما از اشیای همزمان کننده ی جدا از هم در هر lock ، Pulse و Wait استفاده کنیم ، زیرا یک شی از تولید بن بست جلوگیری می کند. بنابراین ، با یک شی قفل گذاری ، همه ی فیلدهای انسداد به عنوان یک فیلد واحد خوانده و نوشته می شوند ، که از خطاهای atomicity ظریف جلوگیری می کند. استفاده از شی همزمان کننده برای اهداف خارج از حوزه لازم ، عقیده ی خوبی است (این کار می تواند با اعلان شی همزمان کننده و همه فیلدهای انسداد ، به عنوان یک شی private انجام شود).

صف تولید کننده / مصرف کننده

یک برنامه Wait/Pulse یک صف تولید کننده / مصرف کننده است – ساختاری که ما قبلاً در فصل 2 با استفاده از AutoResetEvent پیاده سازی کرده بودیم. یک تولید کننده وظایف را وارد صف می کند (معمولاً در نخ اصلی) ، مادامی که یک یا بیشتر از یک مصرف کننده در نخ های کارگر اجرا می شوند و وظایف را یک به یک انجام می دهند.

در این مثال ، ما از یک رشته برای نمایش وظایف استفاده می کنیم. صف وظایفمان سپس شبیه عبارت زیر خواهد بود :

```
Queue<string>  taskQ =  new  Queue<string>();
```

چونکه صف در نخ های متعدد استفاده خواهد شد ، ما باید همه ی دستورات مربوط به خواندن و نوشتن در صف را در یک بلوک lock قرار دهیم. در زیر طریقه وارد شدن وظایف به صف نشان داده شده است :

```
lock (locker)
{
   taskQ.Enqueue ("my task");
   Monitor.PulseAll (locker);   // We're altering a blocking condition
}
```

چونکه ما در حال ویرایش یک شرط انسداد ذاتی هستیم ، ما باید پالس کنیم. ما به جای متد Pulse ، متد PulseAll را فراخوانی می کنیم زیرا قصد داریم تا قابلیت استفاده از مصرف کننده های متعدد را فراهم کنیم. بیشتر از یک نخ ممکن است منتظر بماند.

زمانی که هیچ کاری انجام نمی شود ، ما از نخ های کارگر می خواهیم تا مسدود شوند ، به عبارت دیگر ، وقتی هیچ آیتمی در صف وجود ندارد. از اینرو شرط انسداد ما taskQ.Count==0 می باشد. در زیر دستور Wait ای قرار دارد که دقیقاً همین کار را انجام می دهد :

```
lock (locker)
  while (taskQ.Count == 0)
    Monitor.Wait (locker);
```

گام بعدی مربوط به نخ کارگر می باشد که وظایف را به منظور اجرا از صف خارج می کند :

```
lock (locker)
  while (taskQ.Count == 0)
    Monitor.Wait (locker);

string task;
lock (locker)
  task = taskQ.Dequeue();
```

درحال این منطق ، thread-safe نیست : الگوی ما بر پایه ی تصمیمی برای خروج اطلاعات کهنه از صف می باشد – که در دستور lock قبلی احراز شده بود. اتفاقی را در نظر بگیرید هنگامی که ما دو نخ مصرف کننده را به طور همزمان اجرا می کنیم ، با یک آیتم که هم اکنون در صف وجود دارد. این امکان وجود دارد که هیچ نخی برای مسدود شدن به حلقه ی while وارد نشود – هر دو نخ آیتم موجود در صف را می بینند. اما هر دوی این نخ ها تلاش ناموفقی را برای خروج آیتم از صف در یک زمان دارند ، که یک exception در نمونه ی دوم ایجاد می کند! برای تصحیح این مشکل ، ما به راحتی حوزه عمل دستور lock را کمی بزرگتر می کنیم – تا زمانی که کار ما با صف تمام شود :

```
string task;
lock (locker)
{
  while (taskQ.Count == 0)
    Monitor.Wait (locker);
  task = taskQ.Dequeue();
}
```

(ما بعد از خارج شدن آیتم ها از صف به فراخوانی متد Pulse نیاز نداریم ، زیرا هیچ مصرف کننده ای نمی تواند با این آیتم ها از انسداد بیرون بیاید).

یک بار که وظیفه از صف خارج شد ، هیچ نیازمندی دیگری برای حفظ قفل وجود ندارد. آزاد کردن آن در این نقطه به مصرف کننده اجازه می دهد تا یک وظیفه ی مصرف کننده ی زمان را بدون مسدود کردن غیر ضروری دیگر نخ ها انجام دهد.

در زیر برنامه ی کامل قرار داده شده است. مانند ورژن نوشته شده با AutoResetEvent ما یک وظیفه ی null را برای علامت دادن عمل خروج به مصرف کننده وارد صف می کنیم (بعد از پایان هر وظیفه ی عقب افتاده). چون ما از مصرف کننده های متعدد استفاده می کنیم ، ما باید به ازای هر مصرف کننده یک وظیفه ی null را وارد صف کنیم تا کاملاً کار صف را متوقف کنیم :

***متن استاندارد** Wait/Pulse #2 : صف تولید کننده / مصرف کننده*

```
using System;
using System.Threading;
using System.Collections.Generic;

public class TaskQueue : IDisposable
{
  object locker = new object();
  Thread[] workers;
  Queue<string> taskQ = new Queue<string>();

  public TaskQueue (int workerCount)
  {
    workers = new Thread [workerCount];

    // Create and start a separate thread for each worker
    for (int i = 0; i < workerCount; i++)
     (workers [i] = new Thread (Consume)).Start();
  }

  public void Dispose()
  {
    // Enqueue one null task per worker to make each exit.
    foreach (Thread worker in workers)
      EnqueueTask (null);
    foreach (Thread worker in workers)
      worker.Join();
  }

  public void EnqueueTask (string task)
  {
    lock (locker)
    {
      taskQ.Enqueue (task);
      Monitor.PulseAll (locker);
    }
  }

  void Consume()
  {
    while (true)
    {
      string task;
      lock (locker)
      {
```

```
        while (taskQ.Count == 0) Monitor.Wait (locker);
          task = taskQ.Dequeue();
      }
      if (task == null)
        return;        // This signals our exit
      Console.Write (task);
      Thread.Sleep (1000);            // Simulate time-consuming task
    }
  }
}
```

در زیر متد Main قرار دارد که یک وظیفه را شروع می کند ، دو نخ مصرف کننده ی همزمان را ایجاد می کند و سپس ده وظیفه را به منظور اشتراک بین دو مصرف کننده به صف وارد می کند :

```
static void Main()
{
  using (TaskQueue q = new TaskQueue (2))
  {
    for (int i = 0; i < 10; i++)
      q.EnqueueTask (" Task" + i);

    Console.WriteLine ("Enqueued 10 tasks");
    Console.WriteLine ("Waiting for tasks to complete...");
  }
  // Exiting the using statement runs TaskQueue's Dispose method, which
  // shuts down the consumers, after all outstanding tasks are completed.
  Console.WriteLine ("\r\nAll tasks done!");
}
```

در زیر خروجی اجرای برنامه ی بالا قرار دارد :

```
Enqueued 10 tasks
Waiting for tasks to complete...
 Task1 Task0 (pause...) Task2 Task3 (pause...) Task4 Task5 (pause...)
 Task6 Task7 (pause...) Task8 Task9 (pause...)
All tasks done!
```

مطابق با الگوی ما ، اگر ما متد PulseAll را برداشته و به جای آن متد Wait را به همراه lock toggling قرار دهیم ، ما همین خروجی را خواهیم داشت.

Pulse Economy

با هم تولید کننده را ، وقتی که یک وظیفه را وارد صف می کند دوباره می بینیم :

```
lock (locker)
{
  taskQ.Enqueue (task);
  Monitor.PulseAll (locker);
}
```

اگر بخواهیم دقیق شویم ، ما می توانیم این کار را با انجام پالس در زمانی که امکان آزاد شدن یک نخ کارگر وجود دارد ، صرفه جویی کنیم :

```
lock (locker)
{
  taskQ.Enqueue (task);
  if (taskQ.Count <= workers.Length)
    Monitor.PulseAll (locker);
}
```

به هر حال ما صرفه جویی بسیار اندکی را انجام داده ایم ، زیرا عمل پالسینگ زمانی کمتر از میکروثانیه را تلف می کند ، باعث هیچ overhead ای روی نخ های کارگر نمی شود – زیرا به هر حال آنها آنرا نادیده می گیرند! در یک کد چند نخی ایده مناسب این است که هر منطق غیر ضروری را جمع آوری کنیم : یک خطای متناوب به خاطر یک اشتباه احمقانه هزینه ی سنگینی را برای صرفه جویی یک میکروثانیه می پردازد ، برای نشان دادن این موضوع ، این همه ی چیزی است که این خطا برای ارائه ی یک خطای متناوب "stuck worker" خواهد گرفت ، که بسیار شبیه عمل سرباز زدن از امتحال اولیه است :

```
lock (locker)
{
  taskQ.Enqueue (task);
  if (taskQ.Count < workers.Length)
    Monitor.PulseAll (locker);
}
```

بی خبر انجام دادن عمل پالس ما را از وقوع این مشکل حفاظت می کند.

Pulse یا PulseAll

این مثال با pulse economy بعدی می آید. بعد از وارد کردن یک وظیفه به صف ، ما به جای فراخوانی متد PulseAll می توانیم متد Pulse را فراخوانی کنیم و هیچ اتفاقی نخواهد افتاد.

با هم تفاوت این دو متد را بررسی می کنیم : با متد Pulse ، بیشترین تعداد نخی که می تواند بیدار بماند (و دوباره شرط انسداد حلقه ی while خود را چک کند) یکی می باشد ؛ با متد PulseAll ، همه ی نخ های منتظر بیدار خواهند ماند (و شرط انسداد خود را چک خواهند کرد). اگر ما در حال وارد کردن یک وظیفه به صف باشیم ، فقط یک نخ کارگر می تواند آنرا مدیریت کند ، بنابراین ، ما نیاز به بیدار کردن نخ کارگر با یک متد Pulse هستیم.

در مثال ما ما فقط با دو نخ مصرف کننده کار را شروع کردیم ، بنابراین ما برای دستیابی به نتیجه تعداد اندکی را داریم. اما اگر ما با ده مصرف کننده شروع به کار کنیم ، ما ممکن است بعد از انتخاب Pulse به جای PulseAll سود کمی ببریم. یعنی اگر ما وظایف متعددی را وارد صف کنیم ، ما نیاز داریم تا چندین بار متد Pulse را فراخوانی کنیم. این کار می تواند درون دستور lock انجام گیرد ، همانطور که در زیر آمده است :

```
lock (locker)
{
  taskQ.Enqueue ("task 1");
  taskQ.Enqueue ("task 2");
  Monitor.Pulse (locker);    // "Signal up to two
  Monitor.Pulse (locker);    //  waiting threads."
}
```

هزینه ی یک Pulse حداقل یک stuck worker می باشد. این معمولاً یک مشکل بینابینی را فاش می کند ، زیرا فقط زمانی اتفاق می افتد که یک مصرف کننده در یک حالت **انتظار** قرار دارد.

اگر ارزیابی شرط انسداد به طور غیر معمولی زمانبر باشد ممکن است یک exception تولید کند.

استفاده از زمان های پایان (timeout) متد Wait

ممکن است وقتی یک شرط عدم انسداد اتفاق افتد فراخوانی متد Pulse ، غیر منطقی یا ناممکن به نظر آید. یک مثال ممکن است این باشد که اگر یک شرط انسداد شامل فراخوانی متدی باشد که اطلاعات را از یک جستجوی پایگاه داده دوره ای بدست می آورد. اگر تاخیر یک مشکل نباشد ، راه حل ساده است : می توان یک زمان پایان کار برای متد Wait تعیین کرد ، همانطور که در زیر آمده است:

```
lock (locker) {
  while ( blocking condition )
    Monitor.Wait (locker, timeout);
```

این کار شرط انسداد را مجبور می کند تا دوباره در یک فاصله ی زمانی منظم که همان زمان پایانی است ، همچنین فوراً بعد از دریافت یک پالس چک شود.

همین سیستم درست به خوبی زمانی که پالس به دلیل یک مشکل در برنامه غایب است ، کار می کند! اضافه کردن یک زمان پایان به متدهای Wait در برنامه هایی که ذاتاً دارای پیچیدگی هستند ، ایده ی مناسبی است – به عنوان یک پشتیبان برای مخفی کردن خطاهای عمل پالس. این کار همچنین درجه ای از مصونیت را در صورتی که برنامه توسط افراد دیگری نوشته شود که از Pulse استفاده نمی کنند ، فراهم می آورد!

Race ها و تصدیق

ما علامتی در یک نخ به تعداد 5 بار می خواهیم :

```
class Race
{
  static object locker = new object();
  static bool go;

  static void Main()
  {
    new Thread (SaySomething).Start();

    for (int i = 0; i < 5; i++)
    {
       lock (locker) { go = true; Monitor.Pulse (locker); }
     }
  }

  static void SaySomething()
  {
    for (int i = 0; i < 5; i++)
    {
      lock (locker)
      {
        while (!go)
          Monitor.Wait (locker);
        go = false;
       }
       Console.WriteLine ("Wassup?");
     }
```

```
  }
}
```

خروجی مورد انتظار :

```
Wassup?
Wassup?
Wassup?
Wassup?
Wassup?
```

خروجی اصلی :

```
Wassup?
(hangs)
```

این برنامه دارای نقص می باشد : حلقه ی for در نخ اصلی در هر 5 تکرارش هر بار که نخ کارگر قفل را نگه نمی دارد ، می تواند خلاص شود (آزادانه حرکت کند). حتی ممکن است قبل از اینکه نخ کارگر شروع به حرکت کند! مثال تولید کننده / مصرف کننده در همین بخش از این مشکل رنج نمی برد زیرا اگر نخ اصلی از نخ کارگر جلو بیفتد ، هر درخواست به راحتی از صف خارج می شود. اما در این موقعیت ، ما به نخ اصلی برای انسداد در هر بار تکرار حلقه نیاز داریم البته اگر نخ کارگر همچنان مشغول به انجام وظیفه ی قبلی خود می باشد.

یک راه حل ساده برای نخ اصلی این است که بعد از هر چرخش تا زمانی که پرچم go به وسیله ی نخ کارگر پاک نشده است ، منتظر بماند. در این صورت نیاز است تا نخ کارگر متد Pulse را بعد از پاک کردن پرچم go فراخوانی کند :

```
class Acknowledged
{
  static object locker = new object();
  static bool go;

  static void Main()
  {
    new Thread (SaySomething).Start();

    for (int i = 0; i < 5; i++)
    {
      lock (locker)
      {
        go = true;
        Monitor.Pulse (locker);
      }
      lock (locker)
      {
        while (go)
          Monitor.Wait (locker);
      }
    }
  }

  static void SaySomething()
  {
    for (int i = 0; i < 5; i++)
```

```
    {
      lock (locker)
      {
        while (!go)
          Monitor.Wait (locker);
        go = false;
        Monitor.Pulse (locker);   // Worker must Pulse
      }
      Console.WriteLine ("Wassup?");
    }
  }
}
```

خروجی برنامه بالا به صورت زیر است :

```
Wassup? (repeated five times)
```

یک ویژگی مهم این چنین برنامه هایی این است که نخ کارگر قبل از اینکه وظیفه زمانبر خود را انجام دهد قفل خود را آزاد می کند (این کار جایی که ما متد Console.WriteLine را فراخوانی کردیم ، اتفاق خواهد افتاد). این کار اطمینان حاصل می کند ، مادامی که نخ کارگر وظیفه ای را که به آن علامت داده شده بود را انجام می دهد ، محرک بی جهت مسدود نمی شود (و فقط زمانی مسدود می شود که نخ کارگر مشغول انجام وظیفه ی قبلی خود می باشد).

در این مثال ، فقط یک نخ (نخ اصلی) به نخ کارگر برای انجام وظایف علامت می دهد. اگر نخ های متعددی – با استفاده از منطقی که نخ اصلی ما به کار برد – می خواستند تا این کار را انجام دهند ، ما کمی دچار مشکل می شدیم. دو نخ که در حال علامت دادن می باشند هر کدام می توانند خطوط کد زیر را پشت سر هم اجرا کنند :

```
lock (locker)
{
  go = true;
  Monitor.Pulse (locker);
}
```

اگر نخ کارگر پرازش اولین علامت را به پایان نرساند ، دومین علامت از بین خواهد رفت. ما برای تکمیل کردن الگوی خود از یک جفت پرچم استفاده می کنیم – یک پرچم که "ready" نام دارد و همچنین پرچم "go". پرچم ready نشان دهنده ی این است که نخ کارگر قابلیت پذیرش وظیفه ی جدید را دارد ؛ پرچم go ، همانند قبل ، ساختاری برای اقدام کردن است. این کار با کاری که ما قبلاً در فصل دو با AutoResetEvents انجام دادیم ، قابل مقایسه است ، به جز اینکه این کار توسعه پذیر تر است. در زیر الگوی شرح داده شده قرار دارد :

متن استاندارد** Wait/Pulse #3 : **علامت دهی دو طرفه

```
public class Acknowledged
{
  object locker = new object();
  bool ready;
  bool go;

  public void NotifyWhenReady()
  {
    lock (locker)
    {
      // Wait if the worker's already busy with a previous job
      while (!ready)
```

```
            Monitor.Wait (locker);
        ready = false;
        go = true;
        Monitor.PulseAll (locker);
      }
    }

    public void AcknowledgedWait()
    {
      // Indicate that we're ready to process a request
      lock (locker)
      {
        ready = true;
        Monitor.Pulse (locker);
      }

      lock (locker)
      {
        while (!go)
          Monitor.Wait (locker);      // Wait for a "go" signal
        go = false;
        Monitor.PulseAll (locker);   // Acknowledge signal
      }

      Console.WriteLine ("Wassup?");           // Perform task
    }
}
```

ما نشان دادن عملکرد الگو ، ما با دو نخ همزمان شروع می کنیم ، هر کدام به نخ کارگر را 5 بار اخطار می دهند. ضمناً نخ اصلی برای 10 اخطار منتظر خواهد ماند :

```
public class Test
{
  static Acknowledged a = new Acknowledged();

  static void Main()
  {
    new Thread (Notify5).Start();     // Run two concurrent
    new Thread (Notify5).Start();     // notifiers...
    Wait10();                         // ... and one waiter.
  }

  static void Notify5()
  {
    for (int i = 0; i < 5; i++)
      a.NotifyWhenReady();
  }

  static void Wait10()
  {
    for (int i = 0; i < 10; i++)
```

```
        a.AcknowledgedWait();
    }
}
```

خروجی کد بالا در به این صورت است :

```
Wassup?
Wassup?
Wassup?
(repeated ten times)
```

در متد Notify ، پرچم ready قبل از دستور lock موجود پاک می کند. این کار ضرورتاً لازم است : این کار از علامت دادن دو اخطاردهنده به طور متناوب بدون دوباره چک کردن پرچم ، جلوگیری می کند. به خاطر سادگی کار ، ما همچنین پرچم go را تنظیم کردیم و متد PulseAll را در همان دستور lock فراخوانی کردیم – اگرچه ما می توانستیم این کار را در دو بلوک lock جدا از هم انجام دهیم و با این کار هیچ اتفاقی هم نمی افتاد.

شبیه سازی Wait Handle ها

شما ممکن است الگوی مثال قبلی را دیده باشید : هر دو حلقه ی انتظار دارای ساختار زیر می باشند :

```
lock (locker)
{
  while (!flag)
    Monitor.Wait (locker);
  flag = false;
 ...
}
```

منظور ما جایی است که پرچم flag با مقدار true تنظیم می شود. این کار عملاً تقلیدی از AutoResetEvent می باشد. اگر ما دستور flag=false را حذف کنیم در واقع ما یک ManualResetEvent را خواهیم داشت. با استفاده از فیلدی از اعداد صحیح ، متدهای Wait و Pulse می تواند تقلیدی از یک Semaphore باشد. در حقیقت تنها Wait Handle ای که ما با متدهای Wait و Pulse نمی توانیم ایجاد کنیم Mutex می باشد ، زیرا عملکرد این ساختار با دستور lock فراهم می شود.

شبیه سازی متدهای استاتیکی که در طول wait handle های متعدد کار کنند ، در بیشتر موقعیت ها کار ساده ای است. هم طراز فراخوانی متد WaitAll میان EventWaitHandle های متعدد چیزی بیشتر از یک شرط انسداد نیست که این شرط همه ی پرچم های مورد استفاده به جای wait handle ها را با هم یکی می کند :

```
lock (locker) {
  while (!flag1 && !flag2 && !flag3...)
    Monitor.Wait (locker);
```

این کار به طور ویژه ای می تواند مفید باشد ، چونکه متد WaitAll در بیشتر موقعیت ها به خاطر مشکلات سیاست COM غیر قابل استفاده است. شبیه سازی WaitAny به راحتی با جایگزینی اپراتور && با اپراتور || قابل دستیابی می باشد.

برای شبیه سازی متد SignalAndWait باید کمی زیرکی بکار برد. بیاد بیاورید که این متد مادامی که برای دیگر نخ ها در یک عمل atomic منتظر می ماند ، به یک handle علامت می داد. ما موقعیتی مشابه با یک تراکنش پایگاه داده ی توزیع شده داریم – ما به یک نیروی دو طرفه نیاز داریم ! فرض کنید ما می خواهیم مادامی که منتظر flagB هستیم ، به flagA علامت بدهیم ، ما باید هر پرچم را به دو قسمت تقسیم کنیم ، کد نتیجه شاید چیزی شبیه این باشد :

```
lock (locker)
{
  flagAphase1 = true;
  Monitor.Pulse (locker);
  while (!flagBphase1)
    Monitor.Wait (locker);

  flagAphase2 = true;
  Monitor.Pulse (locker);
  while (!flagBphase2)
    Monitor.Wait (locker);
}
```

اگر اولین دستور Wait به عنوان متوقف شدن یا پایان یافتن ، یک Exception پرتاب کند ، شاید بخواهیم از یک منطق اضافی "rollback" برای منقبض کردن flagAphase1 استفاده کنیم. این یکی از موقعیت هایی است که استفاده از wait handle ها راحت تر است! در هر حال ، یک Singnal-And-Waiting درست در واقع نیازی غیرمعمول است.

Wait Rendezvous

مانند یک WaitHandle.SignalAndWait که برای کنار هم آوردن (Rendezvous) جفتی از نخ ها می تواند استفاده شود ، متدهای Wait و Pulse نیز می توانند این کار را انجام دهند. در مثال زیر ، ممکن است کسی بگوید که ما دو شی ManualResetEvent را شبیه سازی کرده ایم (به عبارت دیگر ، ما دو پرچم از نوع بولین تعریف کرده ایم!) و سپس عمل signal-and-waiting دو طرفه را با تنظیم یک نخ مادامی که منتظر نخ دیگر می باشد ، انجام داده ایم. در این موقعیت ما به atomicity کامل در عمل signal-and-waiting نیاز نداریم ، بنابراین ما احتیاج به یک "نیروی دو طرفه" را می توانیم حذف کنیم. همچنین ما پرچم خود را با مقدار true تنظیم کردیم و در همان دستور lock متد Wait را فراخوانی کردیم ، عمل rendezvous به درستی کار خواهد کرد :

```
class Rendezvous
{
  static object locker = new object();
  static bool signal1, signal2;

  static void Main()
  {
    // Get each thread to sleep a random amount of time.
    Random r = new Random();
    new Thread (Mate).Start (r.Next (10000));
    Thread.Sleep (r.Next (10000));

    lock (locker)
    {
      signal1 = true;
      Monitor.Pulse (locker);
      while (!signal2)
        Monitor.Wait (locker);
    }
   Console.Write ("Mate! ");
```

```
  }

  // This is called via a ParameterizedThreadStart
  static void Mate (object delay)
  {
    Thread.Sleep ((int) delay);
    lock (locker)
    {
      signal2 = true;
      Monitor.Pulse (locker);
      while (!signal1) Monitor.Wait (locker);
    }
    Console.Write ("Mate! ");
  }
}
```

خروجی برنامه ی بالا در زیر آمده است :

```
Mate! Mate! (almost in unison)
```

متدهای Wait و Pulse در مقابل Wait Handle ها

بدیل اینکه متدهای Wait و Pulse انعطاف پذیر ترین ساختارهای همزمان کننده هستند ، آنها تقریباً در هر موقعیتی می توانند مورد استفاده قرار گیرند. در مقابل Wait Handle ها نیز دارای دو فایده هستند :

- آنها قابلیت کار در طول پردازش های متعدد را دارند
- آنها به سادگی قابل فهم هستند ، و عملکردشان به سختی ناموفق تمام می شود.

به علاوه ، Wait handle ها بسیار قابل استفاده تر هستند زیرا می توان آنها را به عنوان آرگومان به متدها فرستاد. در بخش مربوط به مخزن نخ (thread pooling) این ویژگی به خوبی نشان داده شده است.

از لحاظ کارایی ، اگر کسی از الگوی پیشنهاد شده برای انتظار استفاده کند ، متدهای Wait و Pulse برتری جزیی دارند :

```
lock (locker)
  while ( blocking condition )
    Monitor.Wait (locker);
```

و شرط انسداد در ابتدا مقدار false را به خود می گیرد. تنها overhead ای که اتفاق می افتد در اجرای قفل است (ده ها نانوثانیه) در مقابل فقط چند میکروثانیه ای که برای فراخوانی متد WaitHandle.WaitOne صرف می شود. البته ، این فرض وجود دارد که قفل در هنگام اجرا مقابله نکند ؛ حتی مختصرترین مقابله ی یک قفل برای نادیده گرفتن بعضی چیزها کافی است ؛ مقابله ی قفل مکرر باعث می شود که یک Wait Handle سریعتر کار کند!

البته این موضوع میان CPU های مختلف ، سیستم عامل های مختلف ، ورژن های مختلف CLR و منطق برنامه متفاوت خواهد بود؛ و اینکه در هر موقعیت یک میکروثانیه بعید است که به عنوان یک پیامد و اثر قبل از دستور Wait باشد ، کارایی ممکن است دلیل مشکوکی برای انتخاب بین wait handle ها و متدهای Wait و Pulse باشد.

یک رهنمود معقول می تواند این باشد : جایی که ساختار ویژه ای به طور طبیعی خودش را معطوف به یک کار کرده است ، در این موقعیت از Wait handle ها استفاده کنیم ، وگرنه از متدهای Wait و Pulse استفاده کنیم.

متدهای Suspend و Resume

یک نخ مستقیماً و با فراخوانی متد های Thread.Suspend و Thread.Resumeمی تواند به حال تعویق در آید یا دوباره به کار خود ادامه دهد. این مکانیزم کاملاً از بحث مربوط به مسدود کردن که ما در فصل دو به آن پرداختیم ، جدا است. هر دو سیستم به هم وابسته اند و به طور موازی با هم کار می کنند.

یک نخ می تواند خودش یا دیگر نخ ها را به حالت تعویق در آورد. به طور مختصر فراخوانی متد Suspend در یک نخ ، حالت آن نخ را به SuspendRequested تغییر می دهد ، سپس با رسیدن به نقطه ای امن برای garbage collection به حالت Suspended وارد می شود. از اینجا به بعد این نخ فقط زمانی می تواند به کار خود ادامه دهد که به وسیله ی دیگر نخ ها با استفاده از متد Resume فراخوانی شود. متد Resume فقط می تواند روی نخ های معوق شده کار کند و روی نخ های مسدود شده هیچ تاثیری ندارد.

از .NET 2.0 به بعد ، متدهای Suspend و Resume از رده خارج شدند ، استفاده از آنها به دلیل خطر ذاتی آنها در معوق کردن دلخواهانه ی دیگر نخ ها ، لغو شد. اگر یک نخ قفلی را در یک منبع حیاتی که به حالت تعویق در آمده است ، نگه دارد ، کل برنامه (یا کامپیوتر) به بن بست برخورد می کند. این روش بسیار خطرناک تر از فراخوانی متد Abort می باشد – که به آزاد شدن قفل ها در چنین قفل هایی منجر می شود – حداقل از لحاظ تئوری – با قرار دادن کد در بلوک های finally.

در هر حال ، با این کار فراخوانی متد Suspend در نخ جاری در شرایط امنی انجام می شود – و برای انجام چنین کاری کسی می تواند یک مکانیزم همزمان کننده ی ساده را پیاده سازی کند – با یک نخ کارگر در یک حلقه که وظیفه ای را انجام می دهد و متد Suspend در خود فراخوانی می کند و سپس برای شروع دوباره ی کار به وسیله ی نخ اصلی منتظر می ماند ، البته زمانی که وظیفه ی دیگری آماده ی اجرا است. مشکلی که در امتحان این موضوع است این است که آیا نخ کارگر به حالت تعویق در آمده یا نه. کد زیر را در نظر بگیرید :

```
worker.NextTask = "MowTheLawn";
if ((worker.ThreadState & ThreadState.Suspended) > 0)
  worker.Resume;
else
  // We cannot call Resume as the thread's already running.
  // Signal the worker with a flag instead:
  worker.AnotherTaskAwaits = true;
```

این کد کاملاً thread-unsafe است – نخ ها می توانند در هر نقطه ای از این 5 خط پیش دستی کنند – در زمانی که کدام نخ می تواند به جلو برود و حالت آنرا تغییر می دهد. مادامی که این کد می تواند در این حوزه کار کند ، راه حل بسیار پیچیده تر استفاده از راه دیگری است – استفاده از ساختاری مانند AutoResetEvent یا Monitor.Wait. این موارد استفاده از متدهای Suspend و Resume را بی فایده می کند.

متدهای از رده خارج Suspend و Resume دو حالت دارند – خطرناک و بی فایده !

صرف نظر از نخ (aborting thread)

یک نخ می تواند به وسیله ی متد Abort اجباراً پایان بپذیرد :

```
class Abort
{
  static void Main()
  {
    Thread t = new Thread (delegate() {while(true);});   // Spin forever
    t.Start();
    Thread.Sleep (1000);        // Let it run for a second...
    t.Abort();                  // then abort it.
  }
}
```

نخی که به وسیله ی متد Abort پایان می پذیرد فوراً به حالت AbortRequested وارد می شود. اگر این نخ همانطور که انتظار می رود نابود شود ، این نخ به حالت Stopped می رود. فراخواننده برای این مورد می تواند با فراخوانی متد Join منتظر بماند :

```
class Abort
{
  static void Main()
  {
    Thread t = new Thread (delegate() { while (true); });
    Console.WriteLine (t.ThreadState);     // Unstarted

    t.Start();
    Thread.Sleep (1000);
    Console.WriteLine (t.ThreadState);     // Running

    t.Abort();
    Console.WriteLine (t.ThreadState);     // AbortRequested

    t.Join();
    Console.WriteLine (t.ThreadState);     // Stopped
  }
}
```

در بیشتر موقعیت ها درست زمانی که برنامه در حال اجرای نخ می باشد ، متد Abort یک ThreadAbortException در نخ هدف ایجاد می کند. نخی که به پایان می رسد می تواند مدیریت exception را برعهده بگیرند ، اما exception دوباره بعد از پایان بلوک catch دوباره پرتاب می شود (برای اینکه به نخ تضمین دهد همانطور که انتظار دارد به پایان می رسد). به هر حال می توان از دوباره پرتاب شدن exception با فراخوانی Thread.ResetAbort در داخل بلوک catch جلوگیری کرد. بعد از آن نخ به حالت Running دوباره وارد می شود (و مسلماً این نخ دوباره می تواند با متد Abort فراخوانی کند). در مثال زیر ، نخ کارگر هر بار که متد Abort نمی تواند به درستی اجرا شود از حالت مرگ بازمی گردد :

```
class Terminator {
  static void Main() {
    Thread t = new Thread (Work);
    t.Start();
    Thread.Sleep (1000); t.Abort();
    Thread.Sleep (1000); t.Abort();
    Thread.Sleep (1000); t.Abort();
  }
```

```
  static void Work() {
    while (true) {
      try { while (true); }
      catch (ThreadAbortException) { Thread.ResetAbort(); }
      Console.WriteLine ("I will not die!");
    }
  }
}
```

ThreadAbortException به طور ویژه ای توسط خود برنامه در زمان اجرا کنترل می شود ، به این صورت که این exception اگر به درستی مدیریت نشود ، باعث نابود شدن برنامه نمی شود ، برخلاف همه ی دیگر exception های موجود.

متد Abort روی هر نخی و تقریباً با هر حالتی (Running ، blocked ، Suspended یا Stoped) کار خواهد کرد. به هر حال اگر روی یک نخ معوق شده متد Abort فراخوانی شود ، یک ThreadStateException پرتاب می شود – این بار در فراخوانی نخ – و نخ به پایان رسیده تا زمانی که متعاقباً resume نشود ، دوباره شروع به کار نمی کند. در زیر برنامه ای قرار دارد که یک نخ معوق شده را به پایان می رساند :

```
try
{
  suspendedThread.Abort();
}
catch (ThreadStateException)
{
  suspendedThread.Resume();
}
// Now the suspendedThread will abort.
```

عواقب استفاده از Thread.Abort

فرض کنید یک نخ به پایان رسیده متد ResetAbort را فراخوانی نکند ، کسی ممکن است از این نخ انتظار داشته باشد تا به سرعت نابود شود. اما همین که این اتفاق بیفتد ، با یک واسط خوب ، نخ ممکن است در حالت مرگ بیافتد! در زیر فاکتور هایی قرار دارد که ممکن است رسیدن نخ را به حالت AbortRequested به تاخیر بیاندازد :

- سازنده های کلاس استاتیک هرگز به پایان نمی رسند (abort نمی شوند) – چون کلاس را برای باقی دامنه ی برنامه آسیب دار نمی کند)
- همه ی بلوک های catch/finally محترم شمرده می شوند و هرگز در میانه ی کار به پایان می رسند.
- اگر نخ در هنگام فراخوانی متد Abort در حال اجرای کد مدیریت نشده باشد ، اجرای برنامه تا زمانی که دستورات کد مدیریت شده ی بعدی فرا نرسد ، ادامه پیدا خواهد کرد.

آخرین فاکتور می تواند به طور ویژه ای دردسر ساز باشد ، چون خود NET Framework. اغلب کد مدیریت نشده را فراخوانی می کند و بعضی موقع در آن کد برای مدت طولانی ای باقی می مانند. مثالی از این کار زمانی است که از کلاس شبکه یا پایگاه داده استفاده می کنید. اگر منبع شبکه یا سرور پایگاه داده از بین برود یا عملکرد آن برای پاسخ دادن کند شود ، امکان این وجود دارد که اجرای برنامه در کدهای مدیریت نشده برای دقایقی قرار بگیرد که البته به پیاده سازی کلاس وابسته است. در این موقعیت ها ، مشخصاً کسی نمی خواهد تا با متد Join به نخ پایان یافته متصل شود – حداقل نه بدون زمان پایان (timeout)

پایان دادن به کد خالص .NET. مقدار کمتری مشکل زا است ، البته وقتی که بلوک های try/finally یا دستور using برای تضمین رخ دادن پاکسازی به درستی تشکیل شده باشند ، یک ThreadAbortException می بایست پرتاب شود. با این حال ممکن است کسی داوطلب شود تا کار نامطلوبی را انجام دهد. برای مثال ، کد زیر را در نظر بگیرید :

```
using (StreamWriter w = File.CreateText ("myfile.txt"))
  w.Write ("Abort-Safe?");
```

دستور using در C# در واقع یک میانبر گرامری است ، که در این موقعیت کد بالا به صورت زیر باز می شود :

```
StreamWriter w;
w = File.CreateText ("myfile.txt");
try
{
   w.Write ("Abort-Safe");
}
finally
{
   w.Dispose();
}
```

این امکان برای متد Abort وجود دارد تا بعد از ایجاد شی StreamWriter فراخوانی شود اما باید این کار قبل از شروع بلوک try انجام گیرد. در حقیقت ، با رجوع به کد IL تولید شده ، می توان ملاحظه کرد که این متد می تواند بین کدهای مربوط به ایجاد شی StreamWriter و انتساب شدن آن به w قرار داده شود :

```
IL_0001:  ldstr      "myfile.txt"
IL_0006:  call       class [mscorlib]System.IO.StreamWriter
                [mscorlib]System.IO.File::CreateText(string)
IL_000b:  stloc.0
.try
{
  ...
```

در هر دو شیوه ، متد Dispose در بلوک finally گیر انداخته شده است ، که یک فایل بازشده ی تبعیدی (abandon) را نتیجه می دهد – متعاقباً تا زمانی که برنامه به پایان برسد از ایجاد فایل myfile.txt جلوگیری به عمل می آید.

در واقعیت ، موقعیت این مثال هنوز هم دارای وضعیت بدی است ، زیرا یک Abort به احتمال خیلی زیاد در پیاده سازی File.CreateText رخ خواهد داد.این کار باعث می شود تا یک کد مبهم تولید شود – طوری که انگار ما کد برنامه را در اختیار نداریم. خوشبختانه ، کد .NET. کاملاً هم مبهم نیست : ما دوباره می توانیم در ILDASM حرکت کنیم یا حتی خیلی بهتر از ابزار Reflector استفاده کنیم. و به کد اسمبلی تولید شده توسط .NET. نگاه بیاندازیم ، می بینید که این کد سازنده ی کلاس StreamWriter را فراخوانی کرده ، که دارای منطق زیر می باشد :

```
public StreamWriter (string path, bool append, ...)
{
  ...
  ...
  Stream stream1 = StreamWriter.CreateFile (path, append);
  this.Init (stream1, ...);
}
```

در هیچ جایی از این سازنده بلوک try/catch ای وجود ندارد ، یعنی اگر متد Abort در هر جایی از متد Init فراخوانی شود ، stream ساخته شده نابود می شود (abandon) و هیچ راهی را برای خروج از مدیر فایل حاضر در اختیار ما قرار نمی دهد.

به دلیل اینکه disassemble کردن هر فراخوان CLR مورد نیاز به طور مشهودی نشدنی است ، انجام این کار سؤالی را ایجاد می کند که چطور کسی باید به این سمت رود که یک متد abort-friendly بنویسد. رایج ترین متد حل این مشکل این است که از به پایان رسیدن (abort) دیگر نخ ها جلوگیری کنیم – و علاوه بر آن یک فیلد بولین به کلاس نخ کارگر اضافه کنیم ، که نشان دهنده این است که این نخ باید به پایان برسد. نخ به طور مداوم پرچم را چک می کند و اگر دارای مقدار true بود از آن خارج می شود. همانطور که گفته شد این متد مستقیماً یک exception پرتاب می کند. این کار تضمین می کند مادامی که هر بلوک catch/finally در حال اجراست ، نخ دیگر مورد استفاده نمی تواند باشد – درست شبیه فراخوانی متد Abort از دیگر نخ ها ، به جز اینکه نخ فقط از همان مکان ها ، exception خود را پرتاب می کند :

```
class ProLife
{
  public static void Main()
  {
    RulyWorker w = new RulyWorker();
    Thread t = new Thread (w.Work);
    t.Start();
    Thread.Sleep (500);
    w.Abort();
  }

  public class RulyWorker
  {
    // The volatile keyword ensures abort is not cached by a thread
    volatile bool abort;

    public void Abort() { abort = true; }

    public void Work()
    {
      while (true)
      {
         CheckAbort();
         // Do stuff...
         try     { OtherMethod(); }
         finally  { /* any required cleanup */ }
      }
    }

    void OtherMethod()
    {
      // Do stuff...
      CheckAbort();
    }

    void CheckAbort()
    {
      if (abort)
        Thread.CurrentThread.Abort();
    }
  }
}
```

فراخوانی متد Abort توسط همان نخی که می خواهد به پایان برسد یکی از راه هایی است که استفاده از این متد را امن می کند. راه دیگر برای حفظ امنیت زمانی است که شما می توانید نخی را که می خواهید به پایان برسانید در بخش خاصی از کد تعیین کنید ، که این کار معمولاً با استفاده از مکانیزم های همزمان کننده ای مانند Wait Handle یا Monitor.Wait امکان پذیر است. راه سوم برای فراخوانی متد Abort در یک مسیر امن ، زمانی است که شما متوالیاً دامنه برنامه نخ یا خود process را نابود می کنید.

پایان دادن به دامنه های برنامه (Application Domain)

راه دیگری برای پیاده سازی یک کارگر abort-friendly اجرای نخ در دامنه برنامه ی مربوط به خود می باشد. بعد از فراخوانی متد Abort ، اگر بخواهیم ساده بیان کنیم ، دامنه برنامه را از بین می برد ، با این کار هر منبع که به طور ناصحیحی dispose شده بود را آزاد می کند.

اگر بخواهیم دقیق شویم ، اولین گام – پایان دادن به نخ – لازم نیست ، زیرا وقتی دامنه یک برنامه بار نشده است (unloaded است)، همه ی نخ هایی که در آن دامنه در حال اجرای کد می باشند به طور اتوماتیک به پایان می رسد. به هر حال ، ضرر استناد به این روش این است که اگر نخ های به پایان رسیده در یک مدل زمانبر خارج نشوند (شاید به خاطر قرار داشتن کد در یک بلوک finally ، یا برای دیگر دلایلی که قبلاً بحث شده بود) دامنه ی برنامه بار نخواند شد ، و یک خطای CannotUnloadAppDomainException در متد فراخواننده پرتاب می شود. برای این دلایل ، بهتر است که مستقیماً نخ کارگر را به پایان برسانیم ، و سپس قبل از unload شدن دامنه ی برنامه ، متد Join را با قرار دادن یک زمان پایان (timeout) برای آن فراخوانی کنیم.

در مثال زیر ، نخ یک حلقه ی بی نهایت را وارد می کند ، یک فایل را با استفاده از متد File.CreateText abort-unsafe ایجاد می کند و از فایل خارج می شود. نخ اصلی سپس مکرراً شروع به کار می کند و نخ های کارگر را به پایان می رساند. این کار معمولاً در یک یا دو تکرار به خاطر متد CreateText و پیاده سازی درونی آن ، که در پشت یک مدیر فایل (file handle) باز نابود شده قرار می گیرد ، با شکست مواجه می شود :

```
using System;
using System.IO;
using System.Threading;

class Program
{
  static void Main()
  {
    while (true)
    {
      Thread t = new Thread (Work);
      t.Start();
      Thread.Sleep (100);
      t.Abort();
      Console.WriteLine ("Aborted");
    }
  }

  static void Work()
  {
```

```
    while (true)
      using (StreamWriter w = File.CreateText ("myfile.txt")) { }
  }
}
```

خروجی اجرای برنامه ی بالا به صورت زیر است :

```
Aborted
Aborted
IOException: The process cannot access the file 'myfile.txt' because it
is being used by another process.
```

در زیر برنامه اصلاح شده ی بالا قرار دارد به این صورت که نخ کارگر در دامنه ی برنامه ی خود اجرا می شود ، که بعد از اینکه نخ به پایان رسید ، unload می شود. این کار دائماً و بدون ایجاد هیچ خطایی انجام می گیرد ، زیرا unload کردن دامنه ی برنامه، file handle نابود شده را آزاد می کند:

```
class Program
{
  static void Main (string [] args)
  {
    while (true)
    {
      AppDomain ad = AppDomain.CreateDomain ("worker");
      Thread t = new Thread (delegate() { ad.DoCallBack (Work); });
      t.Start();
      Thread.Sleep (100);
      t.Abort();
      if (!t.Join (2000))
      {
        // Thread won't end - here's where we could take further action,
        // if, indeed, there was anything we could do. Fortunately in
        // this case, we can expect the thread *always* to end.
      }
      AppDomain.Unload (ad);          // Tear down the polluted domain!
      Console.WriteLine ("Aborted");
    }
  }

  static void Work()
  {
    while (true)
      using (StreamWriter w = File.CreateText ("myfile.txt")) { }
  }
}
```

خروجی برنامه به صورت زیر می باشد :

```
Aborted
Aborted
Aborted
Aborted
...
```

ایجاد و نابود کردن یک دامنه ی برنامه در کلاسه بندی جهان فعالیت های threading یک فعالیت نسبتاً زمانبر به حساب می آید که چند میلی ثانیه زمان می برد. چون این کار چیزی است که برای انجام شدن ، نامنظم تر از اتفاقی است که در یک حلقه رخ می دهد! همچنین ، جداکننده ای که در دامنه ی برنامه وجود دارد ، عناصر دیگری را معرفی می کنند که هم می توانند مفید باشند و هم مضر، و به اینکه چه برنامه ی چند نخی ای در حال تنظیم است وابسته است. برای مثال ، در یک برنامه متنی واحد ، اجرای نخ در دامنه های متفاوت برنامه می تواند سود بزرگی را فراهم آورد.

پایان دادن به Process ها

راه دیگری که برای پایان یک نخ وجود دارد این است که پروسس پدر نابود شود. مثالی از این موقعیت زمانی است که صفت IsBackground یک نخ کارگر با مقدار true تنظیم شده باشد ، و نخ اصلی درحالی که نخ کارگر همچنان در حال کار است ، به کار خود پایان دهد. نخ پس زمینه در زنده نگه داشتن برنامه ناتوان است و بنابراین پروسس نابود می شود – نخ پس زمینه را به همراه آن فرض کنید.

وقتی یک نخ به خاطر پروسس والد نابود می شود ، مانند یک مرده متوقف می شود و هیچ بلوک finally ای اجرا نمی شود.

همین موقعیت وقتی کاربری یک برنامه ی unresponsive را توسط Windows Task Manager نابود می کند یا وقتی یک پروسس به وسیله متد Process.Kill نابود می شود ، اتفاق می افتد.