Vlákna v C#

Překlad „Threading in C#“ od Josepha Albahari

Jakub Kottnauer

Z původního seriálu vydaného na Programujte.com

Vlákna v C#

Joseph Albahari

1. vydání

Překlad: Jakub Kottnauer
Korektura: Michal Kobelka, Zdeněk Lehocký, Martin Šimeček

Autorizovaný překlad původního textu „Threading in C#“ z knihy „C# 3.0 in a nutshell“

S dotazy a připomínkami k překladu se můžete obrátit na jakubkottnauer@hotmail.com

Obsah

Předmluva

Tento e-book si klade za cíl seznámit čtenáře s problematikou vláken v jazyce C#. Původně jeho vznik nebyl ani zamýšlen, překládal jsem totiž kapitolu o vláknech z knihy „C# 3.0 in a nutshell“ a napadlo mě, že by kvůli délce textu bylo vhodné upravit ho do nějaké přívětivější a lépe tisknutelné podoby.

Ke konvencím v textu: proměnné, názvy jmenných prostorů, tříd (rozhraní, struktur, …) a poprvé použité cizí výrazy by měly být značeny *kurzívou* a názvy metod **tučně**.

> Označuje upozornění

Jsem otevřen jakýmkoliv návrhům na zlepšení ☺

1. Část I

1.1. Začínáme

1.1.1. Co jsou to vlákna?

Jazyk C# podporuje paralelní spouštění kódu pomocí tzv. *multithreadingu*. Přeloženo do normálního jazyka – můžeme spouštět několik částí kódu najednou, každou část na samostatném vlákně. Představte si, že máte aplikaci, která dělá nějaký složitý a dlouhý výpočet, třeba výpočet čísla pí, nebo renderování. Co se stane, pokud takovouhle akci vyvoláte normálně? Aplikace se zasekne, ale jen zdánlivě, ve skutečnosti poběží operace na pozadí, ale zabere pro sebe celé vlákno, takže se aplikace jeví jako zamrzlá – nemůžete kliknout na žádné tlačítko na formuláři. Pokud ale vytvoříte pro výpočet zvláštní vlákno, výpočet bude probíhat na něm a celé jedno vlákno zbude pro zbytek aplikace. Díky tomu pak můžete aplikací ovlivňovat dění na druhém vlákně – pozastavovat výpočet, přidávat nové hodnoty, vykreslovat průběh do nějakého grafu apod.

C# má ve výchozím stavu jedno vlákno, které pro nás vytvoří běhové prostředí CLR, označované jako primární (nebo hlavní) vlákno, na něm běží každá aplikace, ať už je to Hello World nebo třeba e-mailový klient. Pro vytvoření nového vlákna musíme importovat jmenné prostory *System* a *System.Threading*. Základní představu, co to vůbec vlákno je, už máme, takže si ukážeme nějaký příklad vytvoření nového vlákna.

> Pro použití vláken je nutné importovat namespace *System* a *System.Threading*!

```
class PrvniVlakno
{
    static void Main()
    {
        Thread t = new Thread(NapisY);
        t.Start();                              // Spustí NapisY na novém
vlákně
        while (true) Console.Write("x");    // Bude psát znak "x" na
PRIMÁRNÍM vlákně
    }

    static void NapisY()
    {
        while (true) Console.Write("y");    // Bude psát znak "y" na druhém
vlákně
    }
}
```

Teď nějaké vysvětlení výše uvedeného kódu. Primární vlákno vytvoří nové vlákno pojmenované *t*, na kterém spustí metodu **NapisY**. Zároveň s tím poběží na primárním vláknu vypisování písmena „x“.

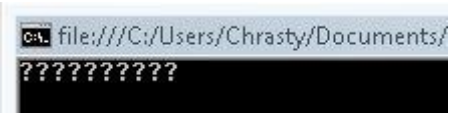
Co by se stalo, kdybychom spustili obě metody na jednom vlákně? Abychom to zjistili, stačí nám jednoduchá úprava kódu.

```
class PrvniVlakno
{
    static void Main()
    {
        while (true) Console.Write("x");
        while (true) Console.Write("y");
    }
}
```

Zkuste tento úryvek zkompilovat, bude se vypisovat jen písmeno „x“. Vypisování písmenka donekonečna je poměrně časově náročná záležitost :-), a proto se vypisování „y“ nikdy nedostane ke slovu.

CLR přiděluje každému vláknu jeho vlastní zásobník paměti, takže vlákna mohou mít své vlastní proměnné. Při zániku vlákna (např. zavoláním metody **Dispose()**) paměť Garbage Collector uvolní a znovu rozdělí pro ostatní vlákna. Jako další ukázku si definujeme jednu metodu s cyklem for (a tedy i lokální proměnnou) a zavoláme ji ze dvou vláken.

```
static void Main()
{
    new Thread(Pis).Start();       // Zavolá Pis() na novém vlákně
    Pis();                          // Zavolá Pis() na primárním vlákně
}

static void Pis()
{
    // Lokální proměnná "cykly" v cyklu
    for (int cykly = 0; cykly < 5; cykly++) Console.Write("?");
}
```



V obou zásobnících se vytvořila jedna kopie proměnné *cykly*, a tak se otazníků vytisklo 10.

Vlákna sdílejí data, pokud mají společný odkaz na stejnou instanci objektu, příklad mluví za vše.

```
class Vlakno
{
    bool hotovo;

    static void Main()
    {
        Vlakno tt = new Vlakno();   // Vytvoření společné instance pro obě
vlákna
        new Thread(tt.Pis).Start();
        tt.Pis();
        Console.ReadKey();
    }
```

```
    // Pis() je teď instanční metodou
    void Pis()
    {
        if (!hotovo) { hotovo = true; Console.WriteLine("Hotovo"); }
    }
}
```

Protože obě vlákna volají metodu **Pis()** ze stejné instance třídy *Vlakno*, sdílejí proměnnou *hotovo*. Díky tomu se „Hotovo“ napíše jen jednou, ne dvakrát. I když to neplatí tak úplně vždy, pokud kód není „*thread safe*“, může dojít k zavolání metody naráz oběma vlákny (co to je „thread safety“ probírám o pár odstavců níže).

Další možností, jak sdílet data, jsou statické proměnné, které jsou sdílené i bez vytvoření instance třídy.

```
class Vlakno
{
    static bool hotovo;     // Statické členy jsou sdíleny všemi vlákny

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

        Console.ReadKey();
    }

    static void Pis()
    {
        if (!hotovo) { hotovo = true; Console.WriteLine("Hotovo"); }
    }
}
```

Oba dva postupy demonstrují jednu klíčovou vlastnost (resp. demonstrují nulovou implementaci té vlastnosti v našem kódu) – tzv. *thread safety* (česky něco jako „vláknová bezpečnost“, zůstaneme raději u originálního názvu) – tu probere v pozdějších kapitolách podrobně, teď jen prozradím, že kousek kódu je „thread safe“, pokud je schopen korektně běžet když je spuštěn více vlákny a hlavně, pokud kód není „thread safe“, existuje malá pravděpodobnost, že k zavolání metody **Pis()** dojde v obou vláknech naráz, takže se i slovo „Hotovo“ vypíše dvakrát. Pokud v metodě **Pis()** prohodíme oba příkazy, šance, že se vypíše „Hotovo“ dvakrát, výrazně stoupne – dokonce hodně nad 50 % (můj skromný odhad činí 60-65 %).

```
static void Pis()
{
    if (!hotovo) { Console.WriteLine("Hotovo"); hotovo = true; }
}
```

Klíčem k nápravě je udělat kód „thread safe“ pomocí tzv. „zámků“ (*locks*). Funguje to tak, že když jedno vlákno operuje s proměnnou, dočasně ji zamkne. Jakmile ukončí práci s ní, znovu ji odemkne. Pokud chce ve stejnou chvíli použít stejnou proměnnou i jiné vlákno, chvíli počká, než mu uvolní místo předchozí vlákno. Takto ošetřený kód už konečně můžeme s čistým svědomím prohlásit za „thread safe“.

```
class Vlakno
{
    static bool hotovo;
    static object zamek = new object();

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

        Console.ReadKey();
    }

    static void Pis()
    {
        lock (zamek)
        {
            if (!hotovo) { Console.WriteLine("Hotovo"); hotovo = true; }
        }
    }
}
```

Různá dočasná pauzování práce vláken jsou nutnou součástí synchronizace jednotlivých vláken. Takovou pauzu můžeme vyvolat i explicitně, ne jen když vlákno narazí na zámek. Pro uspání vlákna slouží metoda **Sleep()**.

```
Thread.Sleep(TimeSpan.FromSeconds(30));  // Přeruší práci vlákna na 30 vteřin
```

Vlákno také může čekat, dokud se neukončí práce jiného vlákna:

```
Thread t = new Thread(Pis); // Pis() je nějaká statická metoda
t.Start();
t.Join();   // Počkat, dokud se vlákno nedokončí
```

Za zmínku stojí, že pauznuté vlákno nespotřebovává systémové zdroje.

1.1.2. Jak vlákna fungují

Multithreading je řízený tzv. *„thread schedulerem“* (plánovač vláken). Zajišťuje všem vláknům nějaký čas jejich spuštění a u vláken, která čekají nebo jsou uspaná, zajišťuje, že nespotřebovávají žádný procesorový čas.

Na jednojádrovém počítači provádí thread scheduler *„time-slicing“* – velkou rychlostí přepíná mezi jednotlivými aktivními vlákny. Pamatujete, jak náš úplně první příklad vypisoval „x“ a „y“ a jednotlivé skupiny nebyly v početně shodných skupinách? Jednou se vypsalo 10 „x“, pak 12 „y“, podruhé to bylo třeba jen 7 „x“ a tak dále. Tyto nerovnoměrnosti jsou dané právě time-slicingem. Ani počítač se netrefí na milisekundu přesně, protože každé vlákno běželo vždy trochu jinou dobu než vlákno druhé. Pro představu jak je time-slicing rychlý – na Windows XP je frekvence přepínání v desetinách milisekund.

Na vícejádrových počítačích (nebo multiprocesorových systémech) funguje multithreading jako mix time-slicingu a čistého běhu (jedno vlákno připadá na jedno jádro). K time-slicingu musí docházet i tak, protože systém musí obsluhovat svá vlastní vlákna, stejně jako vlákna ostatních aplikací.

1.1.3. Vlákna vs. Procesy

Všechna vlákna v jedné aplikaci jsou „uzavřena“ ve společném kontejneru označovaným jako proces. Představte si kabel – balík drátů obalených plastem. Ten obalový plast je proces a jednotlivé dráty jsou vlákna. Proces je jednotka operačního systému, ve kterém běží aplikace.

Vlákna se v ledasčem podobají procesům – například, procesy také podléhají time-slicingu vůči ostatním procesům, jen s tím rozdílem, že procesy jsou naprosto izolované jeden od druhého, zatímco vlákna mezi sebou (uvnitř jedné aplikace) sdílejí haldu (*heap*; jedna z datových struktur). Právě tahle vlastnost dělá vlákny užitečnými – jedno vlákno něco počítá na pozadí a druhé vypisuje výsledky.

1.1.4. Kdy používat vlákna

Budu se opakovat, ale nejčastějším scénám jsou časově náročné operace. Hlavní vlákno ovládá aplikaci, zatímco druhé (pracovní) vlákno zatím vykonává zadanou práci. Na hlavním vlákně nikdy nic složitého nepočítejte, protože ve Windows Forms a WPF aplikacích nemůže aplikace přijímat žádné příkazy z myši ani z klávesnice, pokud je primární vlákno zaměstnané. Navíc systém označí aplikaci „Neodpovídá“ a uživatelé se pak jen bojí, že se aplikace skutečně zasekla.

Další využití najde multithreading u aplikací, které například čekají na odpověď od jiného počítače (databázový server, klient, …). Pokud toto břímě přenecháme pracovnímu vláknu, můžeme implementovat tlačítka jako Cancel, a uživatel je bude moci dokonce použít.

C# aplikace mohou používat multithreading dvěma způsoby. Buď explicitně vytvoříme další vlákna, nebo použijeme některou ze schopností .NET frameworku, které vytvoří další vlákna za nás. Například *BackgroundWorker* (předpřipravená kostra pro pracovní vlákno), threading timery, Web Services nebo ASP.NET aplikace se takto chovají. Jedno vláknový ASP.NET server by nebyl moc užitečný, vždyť zpracovává tolik věcí a přijímá příkazy ze všech stran.

1.1.5. Kdy vlákna nepoužívat

Používání zbytečně velkého počtu vláken může vést k velmi složitému programu. Samotný počet nijak aplikaci nezkomplikuje, vždyť je to jen pár instancí nějaké třídy. Co ale celou věc komplikuje, jsou jednotlivé interakce mezi vlákny. Skutečně není nic těžkého se v nich ztratit, a co teprve odhalování a opravování následných bugů. Kvůli tomuto používejte vlákna s rozvahou a jen, když jsou skutečně potřeba! Nevýhodou jsou i zvýšené nároky na procesor, které plynou z přepínání vláken.

To by na úvod, myslím, stačilo. V následující kapitole se podíváme na základní použití vláken v praxi.

1.2. Vytváření a startování vláken

1.2.1. Základní principy

Vlákna jsou vytvářena pomocí konstruktoru třídy *Thread* předávajícího delegáta *ThreadStart* – ten označuje metodu, kde by měla začít práce vlákna. Takhle vypadá deklarace delegáta *ThreadStart*:

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

Poté následuje zavolání metody **Start()** na instanci vlákna, tato akce uvede vlákno do provozu. Funguje až do chvíle, kdy zpracuje všechny příkazy, které jsme mu zadali. Když vše dokončí, Garbage Collector ho odklidí a uvolní paměť.

```
class Vlakno
{
    static void Main()
    {
        Thread t = new Thread(new ThreadStart(Pis));
        t.Start();     // Spustí Pis() na novém vlákně
        Pis();         // Zároveň s tím zavolá Pis() i na hlavním vlákně
    }

    static void Pis()
    {
        Console.WriteLine("Ahoj!");
    }
}
```

Tento kód vrátí jako výsledek dvě Ahoj!.

Nebylo by to C#, kdyby se nám celou věc nepokusilo trochu zjednodušit. Můžeme celé ThreadStart vypustit, kompilátor si ho tam umí doplnit sám:

```
static void Main()
{
    Thread t = new Thread (Pis);
    ...
}

static void Pis(){ ... }
```

Další způsob, jak si ušetřit práci, je použití anonymních metod:

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

Vlákna mají vlastnost *IsAlive*, která vrací *true*, pokud bylo vlákno už spuštěno (tedy byla zavolána metoda **Start()**), až do jeho zániku. Vlákno po skončení své činnosti nemůže být restartováno, protože ho, jak už jsem zmínil, Garbage Collector odklidí.

1.2.2. Předávání dat delegátovi ThreadStart

Řekněme, že chcete v příkladu nahoře lépe rozlišit, které „Ahoj!“ napsalo které vlákno, třeba tím, že jedno ze slov napíšeme velkými písmeny. Normálně by šlo by předat nějaký parametr metodě **Pis()**, ale to nemůžeme, protože delegát *ThreadStart* nepřijímá žádné argumenty. Naštěstí má .NET framework další verzi delegáta a tou je *ParametrizedThreadStart*, který přímá argument typu *object*, tak jako v příkladu:

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

Upravený příklad z předchozí kapitoly bude vypadat takto:

```
class ThreadTest
{
    static void Main()
    {
        Thread t = new Thread(Pis);
        t.Start(true);  // == Pis(true)
        Pis(false);
    }

    static void Pis(object velkaPismena)
    {
        bool velka = (bool)velkaPismena;
        Console.WriteLine(velka ? "AHOJ!" : "ahoj!");
    }
}
```

Divíte se, kde je nějaký delegát? Kompilátor si ho opět sám dosadí, pokud totiž předáte parametr volané metodě, automaticky se použije *ParametrizedThreadStart* namísto *ThreadStart*.

Parametr předávávaný delegátovi *ParametrizedThreadStart* přijímá právě jeden parametr typu *object*, při použití ho tedy vždy musíme přetypovat, stejně jako já to udělal s přetypováním na *boolean*.

Další možností, jak vyřešit příklad nahoře, je opět použití anonymních metod.

```
static void Main()
{
    Thread t = new Thread(delegate() { Pis("Ahoj"); });
    t.Start();
}

static void Pis(string text)
{
    Console.WriteLine(text);
}
```

Výhoda tohoto postupu spočívá v tom, že metoda **WriteLine** přijímá libovolný počet argumentů a nejsme omezování jako při použití *ParametrizedThreadStart*.

Do třetice, další způsob předávání dat je přes instanční metody namísto statických metod. Jednotlivé vlastnosti instance pak říkají vláknu, co má dělat.

```
class Vlakna
{
    bool velka;

    static void Main()
    {
        Vlakna instance1 = new Vlakna();
        instance1.velka = true;
        Thread t = new Thread(instance1.Pis);
        t.Start();
        Vlakna instance2 = new Vlakna();
        instance2.Pis();
    }

    void Pis()
    {
        Console.WriteLine(velká ? "AHOJ!" : "ahoj!");
    }
}
```

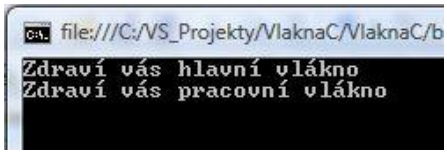
1.2.3. Pojmenovávání vláken

Vlákno můžeme pojmenovat přes vlastnost *Name*. Velmi to usnadňuje debugging (prostě víme, co je které vlákno zač) a s názvy vláken si můžeme hrát i v konzoli.

Jméno vlákna můžeme nastavit kdykoliv se nám zachce, jen musí existovat. Ale pozor, jméno můžeme nastavit jen jednou, jinak dostaneme výjimku!

V následujícím příkladu, protože neběží ve chvíli, kdy upravujeme název vlákna, více než jedno (hlavní) vlákno, můžeme k němu přistoupit přes statickou vlastnost *CurrentThread*.

```
class Pojmenovavani
{
    static void Main()
    {
        Thread.CurrentThread.Name = "hlavní";
        Thread pracovni = new Thread(Pis);
        pracovni.Name = "pracovní";
        pracovni.Start();
        Pis();
        Console.ReadKey();
    }

    static void Pis()
    {
        Console.WriteLine("Zdraví vás " + Thread.CurrentThread.Name + "
vlákno");
    }
}
```



1.2.4. Vlákna běžící v popředí a pozadí

Ve výchozím stavu běží vlákna na popředí, to znamená, že aplikace funguje tak dlouho, dokud alespoň jedno z nich běží. C# umožňuje využití i vláken běžících na pozadí – pokud vypneme všechna vlákna na popředí, aplikace se vypne, i když nějaká vlákna v pozadí ještě fungují.

> Změna vlákna z popředí na pozadí nezmění jeho prioritu vůči ostatním vláknům, ani potřebný procesorový čas.

Vlákna mají vlastnost *IsBackground*, která, jak jistě tušíte, nastavuje (pokud má hodnotu *true*) vlákno na vlákno běžící v pozadí.

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

Pokud je program spuštěn bez parametrů, pracovní vlákno je ve výchozím stavu – běží na popředí, a zastaví se na **Console.ReadLine()**, kde čeká, až uživatel stiskne klávesu Enter. Mezitím hlavní vlákno pořád běží a aplikace funguje, protože hlavní vlákno je aktivní.

Pokud bychom ale metodě **Main()** předali nějaký parametr, pracovní vlákno by se přepnulo do práce na pozadí a aplikace by se téměř okamžitě ukončila, protože hlavní (které běží v popředí) hned ukončí svoji práci a nebere ohledy na to, že nějaké vlákno na pozadí ještě běží.

Když je vlákno běžící na pozadí ukončeno takhle „násilně“, přeskočí se v něm i všechny případné bloky „finally“. Toto chování je nežádoucí (proč bychom nějaké finally vůbec psali, kdybychom ho chtěli přeskakovat), a proto je dobré navyknout si počkat vždy než vlákna na pozadí ukončí svou práci a do té doby práci vláken v popředí pozastavit, třeba pomocí **Thread.Join** (viz předešlá kapitola).

Nastavovat pracovní vlákna jako vlákna běžící na pozadí je výhodné v tom, že máme snadnou kontrolu nad vypínáním aplikace. Představme si opak – vlákno v popředí, které samo při vypnutí aplikace (tedy vypnutí hlavního vlákna) nezemře. Taková aplikace sice zmizí ve Správci úloh ze záložky Aplikace, ale pořád bude její proces aktivní na záložce Procesy. Dokud sám uživatel neukončí na záložce Procesy daný proces, bude běžet a spotřebovávat systémové zdroje.

> Nejčastějším zdrojem problémů vypínaných aplikací jsou zapomenutá vlákna běžící na popředí!

1.2.5. Priorita vláken

Vlastnost vláken zvaná *Priority* určuje, kolik dané vlákno dostane času na vykonání své činnosti. Vzpomínáte na minulou kapitolu, kde jsem se zmiňoval, že CLR přepíná mezi vlákny každou přibližně desetinu milisekundy? Právě vlastností *Priority* se dá tato hodnota mírně upravit.

Tato vlastnost je udělaná jako výčet (typ *enum*) hodnot *Lowest*, *BelowNormal*, *Normal*, *AboveNormal* a *High* (seřazeno od nejnižší priority po nejvyšší). Nastavená hodnota se projeví jen tehdy, pokud je zároveň spuštěno více vláken.

1.2.6. Ošetřování výjimek

Jakékoliv „obecné“ *try/catch/finally* bloky nemají žádný význam, pokud je nové vlákno spuštěné, běží totiž na jiné úrovni a bloky, jako v příkladu níže, bude ignorovat.

```
public static void Main()
{
    try
    {
        new Thread(Pis).Start();
    }
    catch (Exception ex)
    {
        // Sem se ani nikdy nedostaneme!
        Console.WriteLine("Výjimka!");
    }
}

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

Ke bloku *catch* ani nedojde, takže ani *try* by tam nemuselo být. Výsledkem bude nová vlákno s neošetřenou výjimkou *NullReferenceException*. Řešením je napsání těchto bloků zvlášť pro každou metodu, kterou nové vlákno spouští.

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

static void Pis()
{
    try
    {
        ...
        throw null;       // tuhle výjimku to už zachytí
        ...
    }
    catch (Exception ex)
    {
        //Nějaké ošetření výjimky…
        ...
    }
}
```

Od .NET frameworku 2.0 výše, jakákoliv neošetřená výjimka na vlákně shodí celou aplikaci, takže ignorovat je není způsob jak daný problém vyřešit. Bloky *try/catch* musí být v každé metodě (abychom 100% zamezili pádům), což při větším počtu metod začíná být skutečně nepraktické. Jste Windows Forms programátor a používáte časté „globální“ zachycování výjimek?

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

    static void Osetreni(object sender, ThreadExceptionEventArgs e)
    {
        // Zachycení, zapsaní, ošetření výjimky…
    }
}
```

Událost **Application.ThreadException** se zavolá, když naposled volaný kód (jako odpověď na nějakou Windows zprávu) vytvoří výjimku. Toto řešení sice funguje skvěle, ale dává nám falešný pocit bezpečí. Chyby vytvořené pracovními vlákny totiž ani *ThreadException* nezachytí. Naštěstí máme k dispozici low-level řešení – **AppDomain.UnhandledException**. K zavolání dojde kdykoliv, kdy dojde na jakémkoliv vlákně k chybě, v jakémkoliv typu aplikace (ať už s UI nebo bez něj). Ovšem nedoporučuji používat tuto událost jako primární pro zachycení výjimek, použijte ji spíš jako poslední záchranu před pádem aplikace.

To je k základům a tím pádem i první části e-booku vše. V následující části se budeme zabývat základními synchronizačními konstrukcemi.

2. Část II

2.1. Základy synchronizace

Synchronizace, jak název napovídá, slouží ke zkoordinování práce jednotlivých vláken. Následuje několik tabulek, které popisují jednotlivé prostředky k synchronizaci.

2.1.1. Přehled synchronizačních konstrukcí

Jednoduché blokovací metody

Konstrukce	Účel
Sleep	Uspí vlákno na zadaný čas
Join	Počká, než jiné vlákno dodělá svou práci

„Zamykací" konstrukce (locks)

Konstrukce	Účel	Ovlivňuje ostatní vlákna?	Rychlost
lock	Zajistí, že jen jedno vlákno může v jeden okamžik přistoupit k označenému resource souboru nebo části kódu	Ne	Rychlé
Mutex	Viz výše, navíc může zabránit před spuštěním více instancí aplikace	Ano	Střední
Semaphore	Určí, kolik vláken může přistupovat v jeden okamžik k resource nebo části kódu	Ano	Střední

Signalizační konstrukce

Konstrukce	Účel	Ovlivňuje ostatní vlákna?	Rychlost
EventWaitHandle	Přikáže vláknu počkat, dokud nedostane signál od jiného vlákna	Ano	Střední
Wait a Pulse	Vlákno počká, dokud není námi definovaná podmínka splněna	Ne	Střední

Neblokující konstrukce

Konstrukce	Účel	Ovlivňuje ostatní	Rychlost

		vlákna?	
Interlocked	K provedení jednoduchých neblokovacích operací	Ano	Velmi rychlé
volatile	K povolení přístupu k proměnným mimo zámek (lock)	Ano	Velmi rychlé

2.1.2. Blokování

Pokud vlákno čeká, nebo je jeho práce zapauzována následkem některé z výše uvedených konstrukcí, říkáme o něm, že je zablokované. Jakmile je vlákno zablokováno, uvolní se všechny požadované CPU prostředky, do vlastnosti *ThreadState* se uloží hodnota *WaitSleepJoin* a v tomto stavu zůstává, dokud není odblokováno. K odblokování může dojít celkem čtyřmi způsoby (nepočítám vypnutí PC):

- dojde ke splnění blokovací podmínky
- vypršením času, po který má být vlákno blokováno
- přerušením pomocí **Thread.Interrupt**
- zrušením blokování pomocí **Thread.Abort**

2.1.3. Uspávání

Párkrát během seriálu jsem použil pojem „uspávání vlákna“, což je zablokování vlákna na zadaný čas pomocí metody Thread.Sleep (nebo do zavolání Thread.Interrupt).

```
static void Main()
{
    Thread.Sleep(0);                        // vypustí jeden time-slice
    Thread.Sleep(1000);                     // uspí na 1000 ms
    Thread.Sleep(TimeSpan.FromHours(1));    // uspí na 1 hodinu
    Thread.Sleep(Timeout.Infinite);
    // uspí vlákno na nekonečně dlouho dobu (do zavolání Thread.Interrupt)
}
```

Třída *Thread* poskytuje i jednu spíše zajímavost. Tou je metoda **SpinWait()**, která po zavolání neuvolní prostředky CPU, ale naopak ho uzavře do cyklu na zadaný počet iterací. Padesát iterací odpovídá zhruba jedné mikrosekundě (opravdu jen zhruba, závisí to totiž na rychlost a vytížení procesoru). Říkejme pracovně takto zaměstnanému vláknu „zacyklené vlákno“ (oficiální česká terminologie neexistuje, tenhle název je čistě můj výmysl). **SpinWait()** není blokovací metoda, protože zacyklené vlákno nemá hodnotu *WaitSleepJoin* ve vlastnosti *ThreadState*, ani nemůže být přerušena pomocí metody **Interrupt()**. Metoda **SpinWait()** se využívá dost vzácně, její účel je při čekání na data, která mají přijít v horizontu mikrosekund. Takové čekání pomocí **Sleep()** je pak zbytečně náročné. Tento postup má ale smysl pouze u vícejádrových systémů, u jednojádrových se totiž rychleji ukončí time-slice pro aktuální vlákno a tím se ukončí i aktivita **SpinWait()**. Metodě popsané v tomto odstavci se také říká spinning.

2.1.4. Blokování vs. Spinning

Vlákno můžeme zacyklit i známým „trikem“ s while:

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

Tento postup je ale zbytečně náročný na procesor. CLR i operační systém jednoduše pořád dokola kontrolují hodnotu proměnné *proceed*. Úspornější variantou je takový hybrid mezi blokováním a spinningem:

```
while (!proceed) Thread.Sleep (x);
```

Čím větší má proměnná *x* hodnotu, tím je toto úspornější, protože se vlákno uspí a až po uplynutí času x se stav proměnné *proceed* zkontroluje znovu. Cokoliv nad 20 ms je už zase zbytečně přehnané, pokud není podmínka pro cyklus while zvlášť složitá, protože za těch 20 ms už se procesor uvolní pro další iteraci.

2.1.5. Metoda Join()

Dalším z mnoha postupů pro blokování je **Join()** metoda. Po zavolání zablokuje práci aktuálního vlákna, než jiné vlákno dodělá svojí práci. „Zneužiju“ tenhle příklad pro demonstraci lambda výrazů (když už máme ten .NET Framework 3.5).

```
class JoinDemo
{
    static void Main()
    {
        Thread t = new Thread(() => Console.ReadLine());
        t.Start();
        t.Join();     // Čekat, dokud vlákno 't' nedokončí práci
        Console.WriteLine("ReadLine vlákna 't' hotov");
    }
}
```

Tento kód je ekvivalentem k tomuto:

```
class JoinDemo
{
    static void Main()
    {
        Thread t = new Thread(delegate() { Console.ReadLine(); });
        t.Start();
        t.Join();     // Čekat, dokud vlákno 't' nedokončí práci
        Console.WriteLine("ReadLine vlákna 't' hotov");
    }
}
```

Join() přijímá jeden parametr typu *TimeSpan* v milisekundách. Pokud vyprší čas dříve, než se ukončí práce zadaného vlákna, metoda vrátí *false*. S využitím tohoto parametru funguje metoda **Join()** podobně jako **Sleep()**:

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

A další kapitolu máme za sebou, v té následující probereme podrobněji zamykání (locking) a thread safety.

2.2. Locking a thread safety

2.2.1. Locking

Jak už bylo řečeno v úvodním díle seriálu, zamykání zajišťuje exkluzivní přístup k dané části kódu, tedy, že v jednu chvíli může k vymezenému kódu přistupovat jen jedno vlákno. Na začátek tohoto dílu si tuto látku trochu zopakujeme, ať máme na co navazovat. Následuje ukázka, jak by se to dělat nemělo:

```
class ThreadUnsafe
{
    static int val1, val2;

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

Tento postup není thread-safe, pokud by totiž došlo k zavolání **Go()** oběma vlákny naráz, mohlo by dojít k dělení nulou, a to jak známo, není přípustné ani v matematice, ani v programování (program by vyhodil výjimku). Jedno vlákno by totiž mohlo nastavit proměnnou *val2* na nulu zrovna ve chvíli, kdy by se druhá vlákna nacházelo někde mezi *if* a **Console.WriteLine()**. Řešení je nasnadě: použití lock konstrukce.

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

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

V jednu chvíli může udržet kus kódu zamčený jen jedno vlákno a než ho odemkne, všechna vlákna, která se pokusí ke kódu přistoupit, jsou zablokována a řadí se do fronty v pořadí, ve kterém přišla. V příkladu nahoře chráníme obsah metody **Go()**, a tím pádem i obsah proměnných *val1* a *val2*.

Vláknu, které čeká v této frontě, se do vlastnosti *ThreadState* (podrobně probereme v příštím díle) uloží hodnota *WaitSleepJoin*. Příště si také řekneme, že vlákno v tomto stavu můžeme násilně přerušit pomocí metod **Interrupt()** a **Abort()**.

Klíčové slovo *lock* je ve skutečnosti jen taková zkratka (C# nám zase jednou ulehčuje život) pro volání metod **Monitor.Enter()** a **Monitor.Exit()** uvnitř *try-finally* bloku. Ve skutečnosti vidí kód metody **Go()** z posledního příkladu nějak takhle:

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

Volání **Monitor.Exit()** bez předchozího zavolání **Monitor.Enter()** vyhodí výjimku.

Třída Monitor poskytuje i metodu podobnou metodě **Enter()**. Tou je **TryEnter()**, která přijímá jeden parametr v milisekundách, nebo typu *TimeSpan*. Metoda pak vrátí *true*, pokud uzamknutí proběhlo úspěšně v zadaném čase, v opačném případě vrátí false. Ale pozor, tato metoda neuzamyká kód, jen zkouší, jestli je to možné! Originální, nepřetížená verze **TryEnter()** nepřijímá žádný parametr – jen zkusí, jestli by náhodou daný kód nešel uzamknout ve chvíli, kdy dojde k zavolání **TryEnter()**.

2.2.1.1. Synchronizační objekt

Jako synchronizační objekt označujeme něco, podle čeho můžeme synchronizovat práci vláken. Za tímto účelem nám poslouží jakákoliv „věc“ viditelná oběma vlákny. Jediná podmínka je, že musí být referenčního typu. Doporučuje se, aby byl subjekt chráněný proti vnějšímu přepsání, například pomocí modifikátoru *private*.

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

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

Příklad dokazuje, že místo obecného *object* můžeme jako locker využít i instanci třídy *List*.

2.2.1.2. Vložené zamykání

Vlákno může opakovaně uzamknout jeden objekt, buď několikanásobným zavoláním **Monitor.Enter()** anebo přes vložený zámek (*nested lock*). K odemknutí dojde až tehdy, kdy zavoláme **Monitor.Exit()** tolikrát, kolikrát jsme předtím zavolali **Monitor.Enter();**, nebo až se odemkne ten nejvíc vnější zámek.

```
static object x = new object();
static void Main()
{
    lock (x)
    {
        Console.WriteLine("Uzamčeno");
        Vlozka();
        Console.WriteLine("Pořád uzamčeno");
    }
    //Teprve zde dojde k odemknutí
}

static void Vlozka()
{
    lock (x)
    {...}
    //K odemknutí nedojde, tohle není vnější zámek
}
```

2.2.1.3. Kdy zamykat?

Podle nepsaného pravidla by každá proměnná (pole, cokoliv, …) přístupná více jak jedním vláknem měla být „pod zámkem“, když s ní pracujeme. Z následujícího příkladu byste už měli poznat, že to není úplně ideální řešení (ve skutečnosti ideální není vůbec):

```
class ThreadUnsafe
{
    static int x;
    static void Inkrementace() { x++; }
    static void Prirazeni() { x = 123; }
}
```

Po jednoduché úpravě bude tento kód thread-safe jak má být:

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

    static void Inkrementace() { lock (zamek) x++; }
    static void Prirazeni() { lock (zamek) x = 123; }
}
```

Poznámka pro zvídavé – jako alternativa k zamykání existuje neblokující konstrukce, vhodná pro primitivní úlohy. Probereme ji v jedné z posledních kapitol, protože celá problematika není úplně nejjednodušší. Zatím můžete zkusit Google a klíčová slova „thread atomicity“ a „interlocked“, případně si stránku nalistovat.

2.2.1.4. A co výkon?

Zamykání samo o sobě je velmi rychlou akcí, probíhá v desetinách nanosekund. Pokud dojde k zablokování vlákna, už se celá akce zpomalí na mikrosekundy až milisekundy. Ale i toto zpomalení přece stojí za to, mít stabilní aplikaci.

Může nám ale přinést i totální kolaps aplikace, pokud ho nesprávně použijeme. Existují tři takové scénáře: „*Impoverished concurrency*“, „*deadlocks*“ a „*lock race*“. Pod překlady si asi málokdo dokáže něco představit, tak je ani nebudu zmiňovat.

- „Impoverished concurrency“ nastává, když je uzamčeno zbytečně moc kódu, a vlákna se tím pádem blokují na delší dobu, než je nutné.
- „Deadlock“ je, když jsou dvě vlákna uzamčená navzájem, a tak ani jedno nemůže pokračovat v práci.
- „Lock race“ nastane, když dvě vlákna „závodí“ o to, které dřív uzamkne nějaký kód. Pokud se to ale povede nevhodnému vláknu, může tím narušit běh zbytku aplikace.

2.2.2. Thread-safety

Jako thread-safe označujeme kód, u kterého nemůže dojít k nepředvídatelnému chování. Dosáhneme toho hlavně zamykáním a omezením vzájemné interakce mezi vlákny na minimum. Metoda, která je thread-safe ve všech ohledech, se označuje jako „*reentrant*“ (znovuvstupující).

„Obecné“ typy, jako různé proměnné, pole, vlastnosti, …, jsou málokdy thread-safe, kvůli jejich velkému množství (pokud se bavíme o nějaké velké aplikaci). Dalším důvodem je pořád dokola omílaný výkon. Mít vše thread-safe by bylo hezké, ale v praxi tedy špatně proveditelné (sice bychom mohli mít všechno uzamčené ve „velkých“ zámcích, ale zase narážíme na ten výkon…), proto se setkáme thread-safe kódem spíše jen na rizikových místech.

2.2.2.1. Thread-safety a .NET typy

Většina datových typů, kromě primitivních (tedy kromě těch nejzákladnějších typů), nejsou thread-safe. Jasným příkladem thread-unsafe typů jsou kolekce všeho druhu; ukažme si třeba kolekci *List*:

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

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

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

        string[] prvky;
        lock (list) prvky = list.ToArray();
        foreach (string s in prvky) Console.WriteLine(s);
    }
}
```

V tomto případě zamykáme objektem „*list*“ samotným, což je v podobně snadném scénáři dostačující. Pokud bychom ale měli dvě tyto kolekce a byly by nějak provázané, nejspíš bychom museli použít nějaký nesouvisející objekt jako zámek.

Procházení .NET kolekcemi je také thread-unsafe – dojde k vytvoření výjimky, pokud nějaké vlákno kolekci upraví, zatímco druhé vlákno touto kolekcí prochází. Tentokrát bych ale spíše než použití lockingu zkopíroval prvky kolekce do nějakého pole (během kopírování je ale nutné originální kolekci po dobu kopírování locknout) a procházel tou kopií, originál ať si klidně ostatní vlákna upravují, jak se jim zachce.

Teď něco na zamyšlení. Představte si, že by třída *List* byla thread-safe. Pomohlo by nám to nějak? Zase tak moc ne. Na vysvětlení si vezmeme následující kód: chceme přidat prvek do naší „thread-safe“ *List* kolekce.

```
if (!Kolekce.Contains (novyPrvek)) Kolekce.Add (novyPrvek);
```

Celý tento kód by musel být pod zámkem, protože ve chvíli, kdy kontrolujeme, jestli není daný prvek už náhodou v kolekci, by jiné vlákno mohlo tentýž prvek přidat a to by mohlo způsobit další nečekané chování. Je tedy vidět, že thread-safe kolekce by byly ve většině případů zbytečné.

Rýpalové by mohli namítnout, proč se zatěžovat se psaním „vestavěné“ thread-safety při psaní vlastních komponent, když se stejně dá „vše“ vyřešit až při jejich použití pomocí kontrukce *lock*.

Nejhorší případ nastane, když máme statické členy v public typu. Takovým příkladem je třeba struktura *DateTime* a jedna z jejích vlastností, *DateTime.Now*. Pokud by dvě vlákna přistoupila k této vlastnosti v jednu chvíli, výstup by mohl být „zkomolený“, nebo by aplikace rovnou skončila výjimkou. Jedinou možností, jak tohle ošetřit z externího kódu, by bylo uzamknout celý typ, tedy použít

```
lock(typeof(DateTime))
```

Je to ale všeobecně považované za špatné programátorské vychování, takže nepoužívat! :-) Naštěstí je ale struktura *DateTime* (stejně jako ostatní) thread-safe, takže k tomuto nikdy nedojde. U všech vašich komponent (a hlavně u těch, které budete veřejně publikovat) byste se měli postarat o to, aby byly thread-safe samy o sobě.

To je v této kapitole vše, v té další nás čekají metody **Interrupt** a **Abort**.

2.3. Interrupt a Abort

Odblokovat vlákno můžeme dvěma způsoby:

- Pomocí metody **Thread.Interrupt**
- Pomocí metody **Thread.Abort**

K zavolání jedné z těchto metod musí dojít v jiném vlákně, než v tom zablokovaném – zablokované vlákno není schopné dělání čehokoliv, je prostě zablokované.

2.3.1. Interrupt

Zavolání metody **Interrupt** na zablokované vlákno násilně ukončí jeho zablokování a zároveň vytvoří výjimku *ThreadInterruptedException*, stejně jako v následujícím příkladu:

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

        t.Start();
        t.Interrupt();
    }
}
```

Tento kód vypíše do konzole text „Násilně odblokováno!“, je tedy vidět, že výjimka *ThreadInterruptedException* je skutečně vytvořena.

Když zavoláme **Interrupt** na vlákno, které není zablokované, tak se účinek této metody pozdrží, dokud k zablokování nedojde. Pak se opět vytvoří výjimka *ThreadInterruptedException* a všechno pokračuje stejně jako v kódu výše. Díky tomuto opatření nemusíme použít tento test:

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

Který by ani nebyl thread-safe kvůli možnému přerušení mezi podmínkou a voláním **Interrupt**.

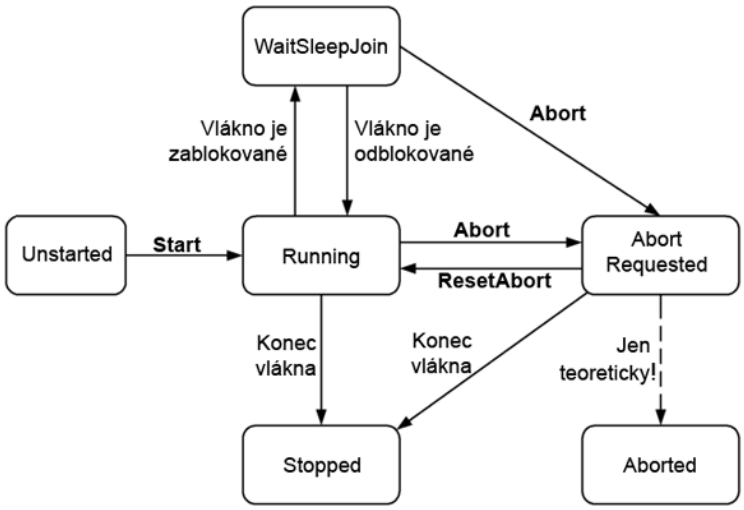
Ovšem přerušení blokace vlákna s sebou nese určitá rizika – pokud na to není kód stavěný, může dojít k narušení běhu aplikace. Bezpečné by to bylo, kdybychom věděli, v jaké fázi své činnosti se vlákno zrovna nachází, a podle toho bychom mohli práci synchronizovat. Že jste to slovo už někde slyšeli? Ano, v předminulém díle byla tabulka různých metod synchronizace a mezi nimi byly i signalizační konstrukce. A ty právě slouží k vyřešení tohoto problému. O těch ale zase příště.

2.3.2. Abort

Tato metoda má podobný účinek jako metoda **Interrupt**, až na pár odlišností. Místo *ThreadInterruptedException* dochází k vytvoření výjimky *ThreadAbortException*. Navíc, tato výjimka je vyhozena ještě jednou na konci *catch* bloku (vlákno se snaží ukončit se, co to jen jde), pokud v tomto bloku nezavoláme metodu **Thread.ResetAbort**. Mezitím má metoda ve vlastnosti *ThreadState* uloženou hodnotu *AbortRequested*.

Největší rozdíl oproti **Interrupt** je ale v tom, jak se tato metoda chová, když ji zavoláme na vlákno, které není zablokované. Zatímco **Interrupt** čeká, než k zablokování dojde, **Abort** vyhodí výjimku v momentě zavolání. Následky mohou být komplikovanější, proto se na ně zaměříme až v jedné z posledních kapitol.

2.4. Stav vlákna (ThreadState)



Vlastnost *ThreadState* slouží ke zjištění aktuálního stavu vlákna. Na této vlastnosti je zajímavé, že má tři „vrstvy“ různých stavů, rozlišené podle klíče jednu od druhé. Na diagramu jsou uvnitř „bublinek“ jednotlivé hodnoty enumerace a tučně metody, které k jednotlivým hodnotám výčtu vedou. Tři vrstvy, o kterých jsem se zmínil před minutkou, jsou rozdělené takto:

1. Jestli vlákno běží, je blokováno, nebo je volána metoda **Abort**
2. Jestli vlákno běží na pozadí, nebo na popředí (*ThreadState.Background*)
3. Podle stavu vlákna vůči metodě **Suspend** (*ThreadState.SuspendRequested* a *Suspended*)

Ve výsledku se dostáváme k velkému množství hodnot, kterých může vlastnost *ThreadState* nabývat. Některé z nich vidíte v diagramu nahoře. Vysvětlím, proč je před hodnotou *Aborted* poznámka „Jen teoreticky!“. Celý výčet možných hodnot totiž obsahuje dvě, které nejsou v současné verzi CLR vůbec implementovány! Jsou jimi *Aborted* a *StopRequested*.

Aby se to celé ještě zkomplikovalo, *ThreadState.Running* má index 0 (indexem teď myslím ten index, který má každá hodnota ve výčtu), takže tento kód by nefungoval:

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

Místo toho musíme použít například negaci výše uvedené podmínky, případně vlastnost *IsAlive*. Ta ale nemusí být vždy to, co chceme, protože ta vrací *true*, i když je vlákno zablokováno (a *false* vrací jen před startem vlákna, nebo až po jeho ukončení).

Za předpokladu, že rozumíte metodám **Suspend** a **Resume** (to nás čeká později), můžete si napsat pomocnou metodu, která vyřadí všechny hodnoty enumerace, kromě hodnot první vrstvy a díky tomu můžete použít jednoduchý test rovnosti. Zda vlákno běží na pozadí můžeme zjistit nezávisle na druhé vrstvě pomocí vlastnosti *IsBackground*, takže nakonec jen první vrstva poskytuje hodnoty, které potřebujeme.

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

ThreadState je neocenitelný pomocník při debuggingu, přesto je jeho využitelnost pro koordinaci více vláken rozporuplná, protože neexistuje žádný mechanismus, pomocí kterého bychom mohli jednoduše zjistit hodnotu *ThreadState* a na základě té informace pak jednat, než by se mezitím ta hodnota změnila.

Můžete si oddychnout, možná trochu složitou problematiku vlastnosti *ThreadState* máme za sebou. Teď se podíváme na signalizační konstrukce *EventWaitHandle*, *Mutex* a *Semaphore*.

2.5. Wait Handles

Všechny tři třídy zmíněné na konci předešlé kapitoly toho mají spolu hodně společného, proto jsem je raději dal do jedné delší kapitoly, délky se nebojte, uvidíte, že to není nic složitého.

Konstrukce *lock* (tedy **Monitor.Enter** a **Monitor.Exit**) je jen jedním z několika způsobů pro synchronizaci vláken. Locking je vhodný pro zajištění exkluzivního přístupu k nějakým resources nebo sekcím kódu, ale existují i způsoby, jak synchronizovat bez exkluzivního přístupu, a těmi se dnes budeme zabývat.

Win32 API má bohatou paletu různých synchronizačních schopností a několik z nich převzal i .NET Framework v podobě tříd *EventEwaitHandle*, *Mutex* a *Semaphore*. Navzájem se od sebe docela liší, například *Mutex* je jen hodně rozšířený *lock*, zatímco *EventWaitHandle* poskytuje unikátní funkcionalitu.

Všechny tři třídy jsou odvozené od abstraktní třídy *WaitHandle* a mají jednu společnou věc. Instancím můžeme přiřadit jméno a podle něj se identifikují mezi ostatními procesy. Co se tím myslí, si vysvětlíme za chvilku.

EventWaitHandle má dvě podtřídy: *AutoResetEvent* a *ManualResetEvent* (pozor, název mate, ani jedna nemá nic společného s událostmi (events), nebo delegáty). Rozdíl mezi těmito dvěma třídami je v tom, že každá volá konstruktor své bázové třídy *EventWaitHandle* s jiným argumentem.

Co se týče výkonu, vše za normálních okolností probíhá v jednotkách mikrosekund.

> *AutoResetEvent* je nejužitečnější ze všech tříd odvozených od *WaitHandle*, společně s konstrukcí lock tvoří základy synchronizace.

2.5.1. AutoResetEvent

AutoResetEvent se svým chováním podobná třeba nějakému turniketu v metru. Strčíte tam lístek a ono to pustí jednoho člověka skrz. „*Auto*“ v názvu třídy odkazuje na skutečnost, že tento turniket se automaticky zavře, jakmile člověk projde. Vláknu přikážeme, aby u turniketu počkalo, pomocí metody **WaitOne**, a aby vložilo pomyslný lístek zase pomocí metody **Set**. Pokud více vláken zavolá **WaitOne**, za turniketem se vytvoří fronta. Jakékoliv nezablokované vlákno může zavolat metodu Set, ale vždy dojde k puštění prvního vlákna ve frontě (nebudeme se přece předbíhat).

Pokud dojde k zavolání **Set** ve chvíli, kdy žádné vlákno ve frontě nečeká, tato metoda počká a spustí se ve chvíli, kdy dojde k zavolání **WaitOne** (tím se jejich efekt okamžitě „vykrátí“ a vlákno projde skrz turniket bez čekání). Ale pozor, tento efekt se nesčítá! I když zavoláte **Set** desetkrát, tak to neznamená, že bude puštěno bez čekání 10 vláken, vždy se to týká jen jednoho a ostatní volňásky „propadnou“.

WaitOne přijímá jeden nepovinný parametr související s časovým limitem. Metoda pak vrátí false, pokud by čas vypršel dřív, než došlo k zavolání **Set**.

Další z metod ve třídě *AutoResetEvent* je metoda **Reset**, to je právě ta, která je automaticky volána a vyresetuje nastavení „turniketu“ na výchozí hodnoty (zavřeno, není vložen žádný lístek).

AutoResetEvent můžeme vytvořit dvěma způsoby. Jeden je přes konstruktor:

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

Přijímá parametr typu *bool*, pokud bychom zadali *true*, okamžitě po vytvoření instance by došlo k zavolání **Set**. Druhým způsobem je vytvoření instance přes bázovou třídu:

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

Kdybychom nastavili *EventResetMode* na *ManualReset*, nevytvořila by se instance třídy *AutoResetEvent*, nýbrž třídy *ManualResetEvent* (probereme níže, ale jak už asi tušíte, je to téměř totéž, jen se metoda **Reset** nevolá automaticky).

Na *WaitHandle* bychom měli zavolat metodu **Close**, pokud ho už nebudeme dále potřebovat, abychom uvolnili systémové prostředky. Ale většinou využíváme funkci „turniketu“ po celou dobu života aplikace, takže **Close** volat nemusíme. V následujícím příkladu si ukážeme funkci *EventWaitHandle* v praxi:

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

    static void Main()
    {
        new Thread(Waiter).Start();
        Thread.Sleep(5000);      // Chvilku počkáme
        wh.Set();                // Pustíme vlákno dál
    }
    static void Waiter()
    {
        Console.WriteLine("Čekám...");
        wh.WaitOne();                          // Čekat na propustku
        Console.WriteLine("Propuštěn!");
    }
}
```

Tento kód vypíše do konzole text „Čekám…“ a o pět vteřin později text „Propuštěn!“, jako důkaz toho, co jsme si řekli výše.

2.5.2. Cross-Process EventWaitHandle

Rychle pro ty, co nevědí, co to *cross-process* znamená. Z názvu se dá odvodit, že je to proces, který dokáže pracovat napříč spuštěnými procesy (samozřejmě jen těmi, které vědí, jak spolu komunikovat). Docílíme toho pomocí konstruktoru *EventWaitHandle*, který umožňuje dát instanci této třídy takové „jméno“. Toto jméno je jen nějaký řetězec a může to být cokoliv, co se nedostane do konfliktu s ostatními procesy (ideální je ve tvaru *MojeSpolečnost.MojeAplikace.Jméno*).

```
EventWaitHandle wh = new EventWaitHandle(false, EventResetMode.AutoReset,
  "MojeSpolecnost.MojeAplikace.NejakeJmeno");
```

Pokud dvě aplikace v sobě spustí tento kód, budou si jejich vlákna moci mezi sebou posílat signály.

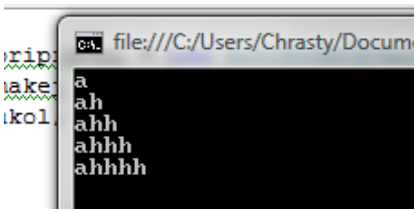
2.5.2.1. Využitelnost

EventWaitHandle využijeme v situacích, kdy budeme chtít provádět úkoly na pozadí, bez nutnosti vytvářet nové vlákno pro každou operaci. Toho můžeme docílit i jednoduchým uzavřením vlákna do smyčky – počká na přidělení úkolu, splní ho, počká na další, splní ho, … Tento postup je poměrně častý, navíc se zamezí riziku v podobě interakce s jiným vláknem a další spotřebou systémových zdrojů.

Musíme se ale nějak rozhodnout, co udělat, když je pracovní vlákno už zaměstnáno jinou činností a do toho dostane další úkol. Logicky druhé vlákno zablokujeme, aby počkalo, než první vlákno dokončí to, co zrovna dělá. Ale jak dáme druhému vláknu vědět, že první vlákno už dokončilo vše, co chtělo? Právě zde přichází ke slovu *AutoResetEvent*. Ukážeme si to na příkladu s jednou proměnnou typu *string* (deklarovanou pomocí klíčového slova *volatile*, které zajišťuje, že obě vlákna vždy uvidí stejnou verzi – probereme v budoucnu):

```
class VyuzitiWaitHandle
{
    static EventWaitHandle pripraven = new AutoResetEvent(false);
    static EventWaitHandle makej = new AutoResetEvent(false);
    static volatile string ukol;

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

        // 5x pošleme signál pracovnímu vláknu
        for (int i = 1; i <= 5; i++)
        {
            pripraven.WaitOne();       // Počkáme, než je pracovní vlákno
připravené
            ukol = "a".PadRight(i, 'h');     // Přidělíme úkol
            makej.Set();     // Přikážeme pracovnímu vláknu, aby pracovalo
        }

        // Řekneme pracovnímu vláknu, aby přestalo pracovat (pomocí
"žádného" úkolu)
        pripraven.WaitOne(); ukol = null; makej.Set();
    }

    static void Pracuj()
    {
        while (true)
        {
            pripraven.Set();     // Indikace toho, že je vlákno připravené
            makej.WaitOne();     // Čekat na propuštění
            if (ukol == null) return;    // Konec
            Console.WriteLine(ukol);
        }
    }
```

Jako výsledek dostaneme to, co je na obrázku. Pro lepší pochopení, jak to celé pracuje, doporučuju dopsat si do kódu různá čekání na stisk kláves atd.



Všimněte si, že práci vlákna ukončuje pomocí *null* úkolu, ale ne pomocí **Interrupt** nebo **Abort**. Je pravda, že by to fungovalo naprosto stejně, jen bychom museli ošetřovat výjimky, které obě metody vytváří, a to je zbytečný kód navíc.

2.5.2.2. Producent/spotřebitel

Častým využitím vláken je, že pracovní vlákno zpracovává úkoly, které stojí za sebou ve frontě, v tzv. frontě producent/spotřebitel (*Producer/consumer queue*). Producent „vytváří“ úkoly a spotřebitelem je to vlákno, které ty úkoly plní (spotřebitelů může být z jedné fronty i více, dobré pro využití potenciálu víceprocesorových systémů).

Princip je podobný jako v předchozí kapitolce u turniketu, jen s tím rozdílem, že volající kód se nezablokuje, pokud je pracovní vlákno už zaměstnané, ale zařadí úkol do fronty a jde „vyrábět“ další úkol.

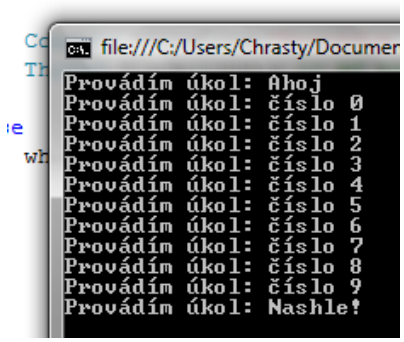
V následujícím příkladu využijeme *AutoResetEvent* pro posílání signálů pracovnímu vláknu, když nemá co dělat (což nastane jen tehdy, když je fronta úkolů prázdná). Fronta úkolů se ukládá do generické kolekce *Queue*, ke které musíme postupovat pod zámkem, aby se zajistila thread-safety. Nakonec celou práci ukončíme předáním *null* úkolu.

```
class ProducerConsumerQueue : IDisposable
{
    EventWaitHandle wh = new AutoResetEvent(false);
    Thread pracovniVlakno;
    object zamek = new object();
    Queue<string> ukoly = new Queue<string>();

    public ProducerConsumerQueue()
    {
        pracovniVlakno = new Thread(Pracuj);
        pracovniVlakno.Start();
    }

    public void ZaraditUkolDoFronty(string task)
    {
        lock (zamek) ukoly.Enqueue(task);
        wh.Set();
    }

    public void Dispose()
    {
        ZaraditUkolDoFronty(null);  // null úkol=konec práce
        pracovniVlakno.Join();   // Počkáme, než pracovní vlákno dodělá to,
co dělá.
        wh.Close(); // Uvolníme systémové prostředky
    }

    void Pracuj()
    {
        while (true)
        {
            string ukol = null;
            lock (zamek)
                if (ukoly.Count > 0)
                {
                    ukol = ukoly.Dequeue();
                    if (ukol == null) return;
                }
            if (ukol != null)
            {
                Console.WriteLine("Provádím úkol: " + ukol);
                Thread.Sleep(1000); // Ať to celé není tak rychlé...
            }
            else
                wh.WaitOne();    // Žádné další úkoly
        }
    }
}
```

Jako poslední dílek skládačky nám chybí metoda **Main**, pomocí které vše otestujeme:

```
class Test
{
    static void Main()
    {
        using (var q = new ProducerConsumerQueue())
        {
            q.ZaraditUkolDoFronty("Ahoj");
            for (int i = 0; i < 10; i++) q.ZaraditUkolDoFronty("číslo " +
i);
            q.ZaraditUkolDoFronty("Nashle!");
        }
        // Díky použití "using" dojde na závěr automaticky k zavolání
        // metody Dispose.
    }
}
```

Vše zkompilujeme a jako výstup dostaneme to, co je na obrázku:



2.5.3. ManualResetEvent

V této kapitole jsem už řekl, že *ManualResetEvent* funguje téměř stejně jako jeho automatický bratr, jen s tím rozdílem, že zde musíme volat metodu **Reset** my, nedělá se to automaticky.

Instance této třídy se někdy používají, chceme-li nějakému jinému vláknu oznámit, že jedno vlákno dokončilo nějakou operaci, nebo říct, že je připravené k práci.

2.5.4. Mutex

Tak jsme dokončili část kapitoly, zabývající se třídou *EventWaitHandle*, teď nás čeká třída *Mutex*.

Samotná funkce *Mutexu* je naprosto zbytečná, protože ji do písmene kopíruje konstrukce *lock*. Jediná výhoda *Mutexu* oproti zamykání je v tom, že dokáže pracovat mezi více procesy, zatímco lock jen v té jedné aplikaci.

> *Mutex* je sám o sobě rychlý, ale *lock* je stokrát rychlejší! Uzamknutí pomocí *Mutex*u trvá pár mikrosekund, ale pomocí *lock*u jsou to desetiny nanosekund, i kvůli výkonu tedy používejte *lock*!

Třída *Mutex*, stejně jako *EventWaitHandle*, obsahuje metodu **WaitOne**, která zajišťuje zámek a všechno blokování. „Odemčení“ dosáhneme pomocí metody **ReleaseMutex**, stejně jako u „locku“, odemknout zámek může jen to vlákno, které ho zamklo.

Typickým využitím *Mutexu* je zajištění toho, že v jednu chvíli může běžet jen jedna instance programu. Následující kód zkuste zkompilovat a výsledný .exe spusťte dvakrát. Uvidíme, že program bude mít námitky.

```
using System;
using System.Threading;

class PustSeJednou
{
    // Ujistěte se, že je název aplikace unikátní, použijte třeba URL adresu vašeho webu
    static Mutex mutex = new Mutex(false, "chrasty.cz PustSeJednou");

    static void Main()
    {
        // 5 vteřin počká, pak ukončí aplikaci
        if (!mutex.WaitOne(TimeSpan.FromSeconds(5), false))
        {
            Console.WriteLine("Jiná instance této aplikace běží! Konec");
            return;
        }
        try
        {
            Console.WriteLine("Aplikace spuštěna - Stiskněte Enter pro ukončení");
            Console.ReadLine();
        }
        finally { mutex.ReleaseMutex(); }
    }
}
```

Dobré, ne? Toto je jedna z věcí, které můžete okamžitě uvést do praxe i bez nějakých hlubších znalostí vláken.

Dobrou funkcí *Mutexu* je, že pokud zapomeneme zámek uvolnit pomocí metody **ReleaseMutex** před vypnutím aplikace, CLR to udělá za nás.

2.5.5. Semaphore

I tuto třídu si přirovnáme k něčemu z reálného světa, třeba ke klubu. Klub má danou kapacitu, kolik dokáže pojmout lidí (podle místa uvnitř). Jakmile je plno, lidé už nemohou dovnitř a před vchodem se tvoří fronta. Jakmile někdo z klubu odejde, jeden člověk ze začátku fronty může dovnitř. Konstruktor třídy *Semaphore* přijímá dva parametry – počet míst v klubu, která jsou momentálně volná, a celkovou kapacitu klubu.

Jakékoliv vlákno může zavolat metodu **Release** na instanci třídy *Semaphore* (v tom se liší od *Mutex* a lockingu, kdy jen vlákno, které blokuje, může propustit blokované vlákno).

Následující příklad vytvoří deset vláken, každé spustí smyčku s metodou **Sleep** uprostřed. A právě třída *Semaphore* zajistí, že **Sleep** nezavolají víc než tři vlákna najednou.

```
class SemaphoreTest
{
    static Semaphore s = new Semaphore(3, 3);   // Dostupná kapacita=3; Celková=3

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

    static void Pracuj()
    {
        while (true)
        {
            s.WaitOne();
            Thread.Sleep(100);   // Maximálně tři vlákna najednou se sem dostanou
            s.Release();
        }
    }
}
```

2.5.6. WaitAny, WaitAll a SignalAndWait

Na závěr tohoto dílu se podíváme na tři metody, které jsou uvedené v nadpisu. Vedle metod **Set** a **WaitOne**, které už znáte, existuje právě tato trojka statických metod ve třídě *WaitHandle*, které slouží pro rozlousknutí složitějších synchronizačních oříšků.

Nejužitečnější je asi metoda **SignalAndWait** – zavolá **WaitOne** na jeden WaitHandle a **Set** na druhý. To můžeme využít na dvojici *EventWaitHandler*ů, abychom zařídili, že se dvě vlákna setkají v jednu chvíli na jednom místě. První vlákno zavolá

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

A druhé zavolá opak:

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

WaitHandle.WaitAny čeká na jakýkoliv ze zadaného pole *Wait Handlerů*; **WaitHandle.WaitAll** čeká na všechny, než podnikne nějakou akci. Máme-li několik turniketů, obě tyto metody vytváří frontu za všemi turnikety. U metody **WaitAny** půjdou lidé skrz první turniket, který se otevře, u **WaitAll** půjdou teprve až se otevřou všechny.

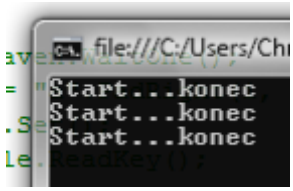
To je k této problematice vše, vrhněme se proto na synchronizační kontexty!

2.6. Synchronizační kontexty

2.6.1. Co je to?

Kromě manuálního zamykání (jiný způsob locking jsme ani nebrali) můžeme zamykat i deklarativně, automaticky. Docílíme toho tím, že třídu odvodíme od třídy *ContextBoundObject* a pak na ni aplikujeme atribut *Synchronization*. Přikážeme tak běhovému prostředí CLR, aby zamykal automaticky:

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

[Synchronization]
public class AutoLock : ContextBoundObject
{
    public void Demo()
    {
        Console.Write("Start...");
        Thread.Sleep(1000); // Nestane se, že by se sem dostala dvě vlákna,
        Console.WriteLine("konec"); // díky automatickému lockingu
    }
}

public class Test
{
    public static void Main()
    {
        AutoLock safeInstance = new AutoLock();
        // Zavoláme 3× metodu Demo
        new Thread(safeInstance.Demo).Start();
        new Thread(safeInstance.Demo).Start();
        safeInstance.Demo();
    }
}
```



CLR zajistí, že jen jedno vlákno může spustit kód uvnitř „*safeInstance*“. Docílí toho vytvořeným svého synchronizačního objektu, který zamkne okolo každého volání metody nebo vlastnosti ze „*safeInstance*“. Oblast, kterou zámek dokáže pokrýt, označujeme za synchronizační kontext.

Jak přesně tohle ale celé funguje? Klíč je v v namespace atributu *Synchronization*, který je *System.Runtime.Remoting.Contexts*. Instanci *ContextBoundObject* si můžeme představit jako „vzdálený“ objekt. Tím je myšleno, že všechny volané metody jsou zachyceny. Zachycení probíhá přes „prostředníka“ – když vytvoříme instanci naší třídy *AutoLock*, CLR vytvoří něco jako prostředníka – objekt se stejnými metodami a vlastnostmi jako *AutoLock*. Právě přes tohoto prostředníka se vykonává automatické zamykání. Ohledně výkonu – automatické zamykání prodlouží volání metody o pár mikrosekund.

Automatické zamykání nemůže být použito na statické členy (vždy je nutná instance, aby se mohl vytvořit prostředník), ani na třídy, které nejsou odvozené od *ContextBoundObject* (příkladem budiž Windows formulář odvozený od třídy *Form*).

Synchronizační kontext může přesáhnout rámec jednoho objektu (instance třídy). Pokud byl synchronizovaný objekt vytvořen z jiné třídy, obě třídy pak sdílejí stejný kontext (jeden velký zámek). Toto chování se dá upravit pomocí nastavení atributu *Synchronization* přes třídu *SynchronizationAttribute* (ta obsahuje několik pojmenovaných „*int*“ konstant, uvedených v tabulce):

Konstanta	**Význam**
NOT_SUPPORTED	Totéž, jako když nepoužijeme atribut vůbec
SUPPORTED	Spojí existující kontext s právě vytvořeným, pokud žádný další neexistuje, nic se nestane
REQUIRED (výchozí)	Spojí existující kontext s právě vytvořeným, pokud žádný další neexistuje, vytvoří se nový
REQUIRES_NEW	Vždy vytvoří nový kontext

Takže pokud přes instanci třídy *SynchronizaceA* vytvoříme instanci třídy *SynchronizaceB*, oba budou mít svůj vlastní synchronizační kontext (pokud je *SynchronizaceB* deklarována jako v příkladu níže).

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

Je logické, že čím větší rozsah kontextu, tím je vše snadnější pro správu, ale jsme více omezení možnostmi. Na druhou stranu – při použití hodně kontextů zase hrozí „deadlocky“ (pro připomenutí – zamezení práce dvou vláken navzájem):

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

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

```
        }
}
```

Protože je každá instance třídy *Deadlock* vytvořená uvnitř třídy *Test* (bez atributu *Synchronization*), každá instance bude mít svůj synchronizační kontext, a tím pádem i svůj lock. Když se dva objekty různě volají navzájem, jako to dělají v příkladu výše, netrvá dlouho a dojde k deadlocku. Na tento „jev“ je nutné dávat si trochu pozor, u automatického lockingu nemusejí být příčiny vždy tak zřejmé.

2.6.2. Reentrancy

S tímto pojmem jsme se setkali ve třetí kapitole, označuje metodu, která je absolutně thread-safe – jedno vlákno ji může zavolat hned po zavolání jiným vláknem bez jakýchkoliv neočekávaných efektů atd. Poměrně často mají pojmy „*thread-safe*“ a „*reentrant*“ stejný význam.

Avšak nedají se úplně zaměnit. Pravá „reentrant“ metoda vznikne, když atribut *Synchronization* rozšíříme o klíčové slovo *true*:

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

Pokud běh aplikace opustí na chvíli ten „svůj“ kontext, dojde k jeho dočasnému zaniknutí. Toto by zabránilo v příkladu nahoře deadlockům! Na druhou stranu, kterékoliv vlákno je pak schopné zavolat kteroukoliv metodu v označené třídě („znovu-vstupovat“ do kontextu; *reentering*), následkem toho ale mohou být další problémy, kterým se celou tu dobu snažíme vyhnout.

> Protože je atribut *[Synchronization(true)]* aplikován na úrovni třídy, každá metoda v označené třídě se tak stává náchylná na „volné“ volání!

Uvedli jsme si několik příkladů, které ukazují některé nevýhody automatického lockingu. Pokud ho použijeme, můžou vyjít na povrch problémy, které by jindy ani nevznikly. Díky tomu je ve složitějších aplikacích výhradně používané manuální zamykání.

Jsme na konci druhé části knihy, týkající se základů synchronizace. Celá třetí část bude praktičtějšího rázu, bude totiž o často používaných třídách a konstrukcích.

3. Část III

3.1. Model apartmentů a WinForms

3.1.1. Apartmenty a jejich význam

Existují tzv. modely apartmentů (oddělení), úzce spjaté s COM (velmi zjednodušeně řečeno je to předchůdce .NET frameworku). Od takovýchto modelů se sice .NET snaží oprostit, ale někdy je můžete potřebovat, hlavně při práci se staršími API (mezi něž se tedy COM řadí). Ve skutečnosti najdou využití častěji, než se může na první pohled zdát. Vždyť WinForms používají z velké části obalené Win32 API!

Apartment je logický „kontejner“ pro vlákna. Existují dva druhy – „single“ a „multi“. První jmenovaný obsahuje vždy jen jedno vlákno (*STA*), druhý obsahuje větší počet vláken (*MTA*).

Kromě samotných vláken obsahují apartmenty i objekty. Když dojde k vytvoření objektu uvnitř apartmentu, zůstane v něm po celou dobu jeho životnosti. V tomto se modely apartmentů podobají synchronizačním kontextům s tím rozdílem, že kontexty neobsahují vlákna. V kontextech může jakékoliv vlákno přistupovat k objektu v kontextu, ale k objektu v apartmentu může přistupovat jen vlákno ze stejného apartmentu.

Opět si vysvětlíme význam apartmentů na příkladu „ze života“. Představte si knihovnu, kde každá kniha reprezentuje objekt. Půjčování knih domů z knihovny není dovoleno – kniha (objekt) v ní zůstane po celou dobu života knihovny. Řekněme, že člověk, který do knihovny vstoupí, je vlákno.

Kdyby knihovna fungovala jako synchronizační kontext, mohl by dovnitř vstoupit jen jeden člověk najednou. Pokud by jich bylo víc, před vchodem by se začala tvořit fronta.

V apartmentové knihovně pracují knihovníci – jeden knihovník pro STA knihovnu, celý tým knihovníků pro MTA. Do knihovny nemůže vstoupit žádný cizí člověk - zákazník, který chce nějakou knihu, musí nejdřív signalizovat knihovníka (tomuto procesu se říká „*marshalling*“). Marshalling probíhá automaticky, ve WinForms funguje tak, že tento mechanismus neustále kontroluje vstupy z klávesnice i myši. Pokud jednotlivé zprávy o těchto událostech přijdou moc rychle za sebou, zařadí se do fronty a vykonají se v pořadí, ve kterém přišly.

3.1.1.1. Nastavení apartmentu

Vláknu je automaticky přiřazen apartment. Výchozí situace je ta, že dostane multi-threaded apartment, pokud explicitně neřekneme, že chceme STA, jako třeba takto:

```
Thread t = new Thread(mydelegate,0);
t.SetApartmentState(ApartmentState.STA);
```

Můžeme bez obtíží přikázat, aby i hlavní vlákno bylo v STA, což se udělá pomocí atributu *[STAThread]*:

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

Slušelo by se napsat pár řádků o využitelnosti apartmentů v praxi. Nemají žádný efekt, pokud s nimi spouštíte čistý .NET kód. Když dva STA kódy zavolají metodu na stejný objekt, nenastane žádný automatický locking, marshalling ani nic podobného. Prostě jako kdybyste spouštěli obyčejný kód. Jen u spouštění unmanaged kódu se jejich síla může projevit.

Typy v namespace *System.Window.Forms* často využívají původní Win32 kód dělaný pro běh v STA. Kvůli tomu by měla mít aplikace využívající WinForms atribut *[STAThread]* u své **Main** metody, jinak by mohlo dojít k pádu aplikace.

3.1.2. Control.Invoke

Ve vícevláknových aplikacích není povolené zavolat metodu nebo vlastnost na ovládací prvek jiným vláknem než tím, které daný prvek vytvořilo. Každý, kdo někdy začal ve svých programech využívat vlákna, se určitě velmi brzo s tímto problémem setkal. Řešením jsou právě metody **Control.Invoke** a **Control.BeginInvoke**, pomocí kterých můžete přesměrovat volání metody z jednoho vlákna na „autorské“ vlákno prvku. Nemůžeme se totiž spoléhat na automatický marshalling zmíněný výše, protože k němu dojde jen tehdy, dostane-li se aplikace do unmanaged kódu. Než k tomu dojde, bude už nejspíše pozdě.

> WPF je podobné Windows Forms v tom, že ovládací prvky jsou přístupné jen z vlákna, které je vytvořilo. Ekvivalent k **Control.Invoke** je ve WPF **Dispatcher.Invoke**.

Dalším perfektním řešením je známý *BackgroundWorker*. Mocná třída, která obaluje pracovní vlákna a mimo jiné volá automaticky podle potřeby **Control.Invoke**.

3.2. BackgroundWorker

BackgroundWorker je pomocná třída v namespace *System.ComponentModel* pro správu pracovních vláken. Poskytuje následující věci:

- Vlastnost *Cancel* pro signalizování vláknu, aby skončilo i bez volání metody **Abort**
- Protokol pro podávání zpráv o průběhu práce, dokončení a zrušení práce
- Implementace rozhraní *IComponent*, které *BackgroundWorkeru* dovoluje, aby se s ním dalo pracovat ve Visual Studio Designeru
- Zachycování výjimek na pracovním vláknu
- Schopnost aktualizovat WinForms (i WPF) ovládací prvky na základně průběhu vlákna

Poslední dva body jsou nejspíš ty nejužitečnější – nemusíte ve svých metodách spouštěných pomocí BW používat *try/catch* bloky a můžete WinForms a WPF upravovat i bez volání **Control.Invoke**.

BW využívá tzv. fond vláken (*thread-pool*), který spravuje vytvořená vlákna a sám ukončuje jejich práci. Z tohoto důvodu byste nikdy neměli na *BackgroundWorker* vlákno volat metodu **Abort**, o to se sám postará fond vláken.

Následují některé kroky, které musíte podstoupit, abyste mohli ve své aplikaci využít *BackgroundWorker*:

- Vytvořit instanci třídy *BackgroundWorker* a vytvořil handler pro událost **DoWork**
- Zavolat metodu **RunWorkerAsync** (nastartuje instanci *BW*)

Jako argument metody RunWorkerAsync můžete zadat cokoliv, přijímá totiž typ *object*. Zadaný argument se následně předá zpracovateli události **DoWork**:

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

    static void bw_DoWork(object sender, DoWorkEventArgs e)
    {
        // Obsah této metody je volán na pracovním vláknu
        Console.WriteLine(e.Argument);  // Vypíše „Zpráva pro DoWork"
        // Další kód…
    }
}
```

Kromě **DoWork** najdete v *BackgroundWorkeru* událost **RunWorkerCompleted**, která se vypálí, když **DoWork** dokončí svojí práci. Zpracování této události není povinné, ovšem většinou je to užitečné, můžete například zpracovat výjimky, které vzniknou během **DoWork**. Navíc, zpracovatel této události může přímo přistupovat k WinForms a WPF prvkům bez explicitního marshallingu, zatímco **DoWork** nemůže.

Pro přidání podpory pro ohlašování pokroku práce musíme udělat následující:

- Nastavit vlastnost *WorkerReportsProgress* na *true*
- Zevnitř **DoWork** handleru volat metodu **ReportProgress** s hodnotou, která určuje procentuální splnění práce
- Vytvořit zpracovatele události **ProgressChanged**, který se bude dotazovat na hodnotu vlastnosti *ProgressPercentage* (v ní je právě uložená hodnota, kterou jsme předali metodě **ReportProgress**)

Kód uvnitř **ProgressChanged** zpracovatele může, stejně jako **RunWorkerCompleted**, volně komunikovat s ovládacími prvky. Toto je místo, které se často využívá k aktualizování *ProgressBaru*.

Přidání podpory pro ukončení práce vlákna zase přidáme takhle:

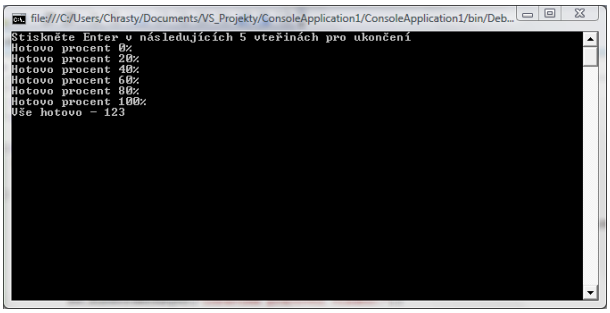
- Nastavte vlastnost *WorkerSupportsCancellation* na *true*

- Zevnitř **DoWork** handleru sledujte stav vlastnosti *CancellationPending*. Pokud je její hodnota *true*, nastavte argument „*e*“ u **DoWork** na *Cancel = true*; (jistě, že můžete nastavit *e.Cancel* na *true* i bez kontrolování *CancellationPending*)
- Zavolejte **CancelAsync** pro ukočení práce

Následující příklad ukazuje vše, co jsme si výše popsali (v kódu jsou využity inicializátory objektů):

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

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

        bw.RunWorkerAsync("Zdravíme pracovní vlákno!");

        Console.WriteLine("Stiskněte Enter v následujících 5 vteřinách pro
ukončení");
        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;    // Tato hodnota se předá do RunWorkerCompleted
    }

    static void bw_RunWorkerCompleted(object sender,
    RunWorkerCompletedEventArgs e)
    {
        if (e.Cancelled)
            Console.WriteLine("Zrušili jste práci");
        else if (e.Error != null)
            Console.WriteLine("Vyskytla se výjimka: " + e.Error);
        else
            Console.WriteLine("Vše hotovo - " + e.Result);      // Hodnota,
kterou jsme předali na konci DoWork
    }
```

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



3.2.1. Odvozování od BackgroundWorker

Třída *BackgroundWorker* není *sealed* (takže od ní můžeme odvozovat další třídy) a poskytuje virtuální metodu **OnDoWork**. Díky tomu, pokud píšeme metodu, jejíž průběh by mohl dlouho trvat (např. načítání nějaké databáze), můžeme vrátit instanci třídy odvozené od *BackgroundWorker*u, která je tím pádem už předpřipravená na asynchronní operace. Jednoduché řešení, které ani nezabere příliš času na implementaci:

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

public class FinancialWorker : BackgroundWorker
{
    public Dictionary<string, int> Result;
    public volatile int Foo, Bar; // I k vysvětlení volatile
    public FinancialWorker() // se jednou dostaneme:-)
    {
        WorkerReportsProgress = true;
        WorkerSupportsCancellation = true;
    }

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

    protected override void OnDoWork(DoWorkEventArgs e)
    {
        ReportProgress(0, "Právě jsem začal pracovat!");

        // ... finished_report indikuje,
        // jestli je ještě co dělat
        while (!finished_report)
        {
            if (CancellationPending)
```

```
            {
                e.Cancel = true;
                return;
            }

            // ... tady by mělo být spočítání průběhu v %
            // uložené jako percentCompleteCalc
            ReportProgress(percentCompleteCalc, "Už to skoro bude...");
        }
        ReportProgress(100, "Hotovo!");
        e.Result = Result;
        // Zpracovat data...
    }
}
```

Když dojde k zavolání metody **GetFinancialTotalsBackground**, dostanete nový objekt typu *FinancialWorker*. Je schopný zpracovávat operace na pozadí, může oznamovat svůj průběh, může být ukončen a je kompatibilní s WinForms i bez (přímého) použití **Control.Invoke**.

A je to! Konečně jsme se dostali k věci, kterou dokážete jistě využít v jakékoliv multi-threaded aplikaci. Teď nás čekají třídy *ReaderWriterLockSlim* (novinka v .NET 3.5) a *ReaderWriterLock*.

3.3. Třídy ReaderWriterLockSlim a ReaderWriterLock

3.3.1. Jak vypadají a k čemu jsou?

Většinou jsou instance různých tříd, typů, … thread-safe pro čtecí operace, ale ne pro zápis nebo aktualizaci obsahu. To platí třeba i u souborů – klidně dvacet vláken najednou ho může číst, ale jen těžko do něj mohou najednou zapisovat. I když si jednoduchý locking většinou s tímto problémem poradí, může být někdy zbytečně omezující, pokud třeba existuje hodně „čtenářů“ obsahu, ale k nějakému zápisu dojde jen občas. Příkladem může být nějaký server s daty, kde se často používaná data cachují do statických proměnných.

> *ReaderWriterLockSlim* je novinka v .NET frameworku 3.5, kde nahrazuje starší třídu *ReaderWriterLock*. „Slim“ se podobá svému předku ve funkcionalitě, ale je rychlejší a ve své podstatě jednodušší. Stará verze měla dokonce několik málo známých bugů, které mohly způsobit nepředvídatelné pády aplikace! Proto budu v tomto textu používat novější *ReaderWriterLockSlim*.

Obě třídy mají dva základní typy locků: „čtecí“ a „zapisovací“ zámek (*read lock*, resp. *write lock*). Write lock zajišťuje vždy exkluzivní přístup k souboru, je tedy určen, překvapivě, k zapisování, zatímco jeden read lock je „kompatibilní“ s ostatními read locky, takže čtení ze souboru není omezené.

Jinak řečeno, vlákno, které momentálně má svůj write lock blokuje všechna ostatní vlákna, která se pokusí získat vlastní write nebo read (!) lock. Ale pokud žádné vlákno zrovna write lock nemá, může získat read lock libovolný počet vláken.

Třída *ReaderWriterLockSlim* má následující metody, které slouží k zamykání a odemykání pomocí read/write locků:

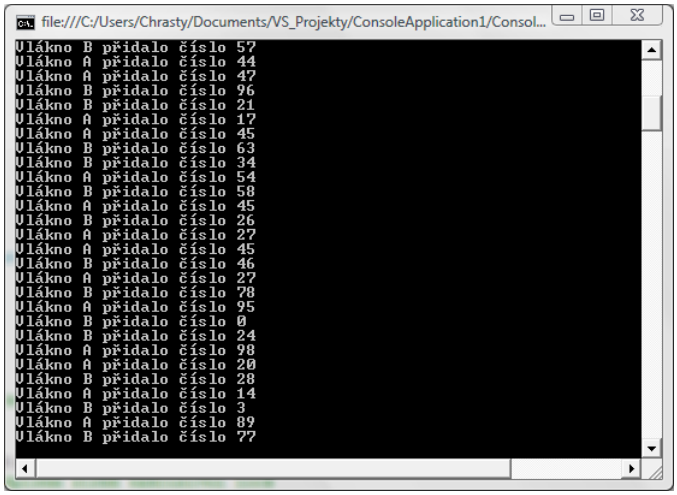
```
public void EnterReadLock();
public void ExitReadLock();
public void EnterWriteLock();
public void ExitWriteLock();
```

Ještě existují „Try“ verze obou „EnterXXX“ metod (například **TryEnterReadLock**), které přijímají číselný argument určující, jak dlouho má vlákno čekat na získání zámku. (Vzpomínáte si ještě na **Monitor.TryEnter**? Ten funguje také tak.) Starý *ReaderWriterLock* má podobné metody, pojmenované „AcquireXXX“ (místo „EnterXXX“) a „ReleaseXXX“ (místo „ExitXXX“).

Následující příklad ukazuje *ReaderWriterLockSlim* v praxi. Tři vlákna neustále procházejí kolekci, zatímco dvě další vlákna každou vteřinou přidají do této kolekce nějaké náhodné číslo. Díky exkluzivnímu přístupu ke kolekci se nestane, že by obě zapisovací vlákna přidala číslo najednou.

```
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)
    {
        // V nekonečné smyčce zapisuje do kolekce
        while (true)
        {
            int newNumber = GetRandNum(100); // Vygenerované číslo
            rw.EnterWriteLock(); // Před zápisem získá exkluzivní lock
            items.Add(newNumber);
            rw.ExitWriteLock(); // "Odemkne" kolekci pro ostatní vlákna
            Console.WriteLine("Vlákno " + threadID + " přidalo číslo " +
newNumber);
            Thread.Sleep(100);
        }
    }
```

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



> Ve skutečném kódu byste ještě měli přidat *try/catch* bloky, abyste zajistili, že zámek bude skutečně odemčen, i kdyby se vyskytla nějaká výjimka.

Právě demonstrovaná třída umožňuje zjišťování mnohem víc informací, než by dovolil obyčejný lock. Přidejte například do metody **Write** tento kód na začátek *while* smyčky:

```
Console.WriteLine("Právě čtou " + rw.CurrentReadCount + " vlákna");
```

Jak už jste asi uhodli, *CurrentReadCount* vrací počet vláken, která jsou momentálně pod read lockem. Většinou bude vypisovat: „Právě čtou 3 vlákna“, protože všechna tři vlákna tráví nejvíce času ve *foreach* smyčce. Kromě této vlastnosti obsahuje třída *ReaderWriterLockSlim* několik dalších:

```
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; }
```

Někdy může být užitečné prohodit read lock za write lock a vytvořit tak jednu atomickou operaci (atomická operace je ta, která nejde přerušit). Například, pokud byste chtěli přidat prvek do

nějaké kolekce, ale jen pod podmínkou, že daný prvek se ještě v kolekci nenachází. Postup by mohl být takovýto:

1. Uzamknout kolekci pod read lockem.
2. Zkontrolovat, jestli už prvek v kolekci je. Pokud ano, uvolnit zámek a zavolat return.
3. Uvolnit zámek.
4. Uzamknout kolekci pod write lockem.
5. Zapsat prvek.

Problém při tomto postupu ale je, že jiné vlákno se může proplížit do této operace, zatímco jsou „odemčené“ všechny zámky a mezitím vložit stejný prvek jako ten, který už máme v plánu přidat (pokud se to stane těsně před uzavřením write locku, prvek se přidá dvakrát!). Třída *ReaderWriterLockSlim* ale naštěstí počítá i s tímto scénářem, a proto poskytuje třetí typ locku – tzv. *upgradeable lock*. Speciální druh zámku, který můžeme později nastavit na write lock. Celá operace je pak pod jedním velkým zámkem, jen se mění jeho povaha. Jiné vlákno tedy nemůže narušit práci. S využitím upgradeable locku bychom postupovali takto:

1. Zavolat metodu EnterUpgradeableReadLock.
2. Provést čtecí operace (kontrola, jestli už prvek v kolekci existuje).
3. Zavolat metodu EnterWriteLock (změní upgradeable lock na write lock).
4. Provést zapisovací operace (přidání prvku do kolekce).
5. Zavolat metodu ExitWriteLock (změní write lock zpátky na upgradeable).
6. Jakékoliv další operace čtecího charakteru.
7. Zavolat metodu ExitUpgradeableReadLock.

V bodu 3 to funguje tak, že read lock se uvolní a vytvoří se zbrusu nový write lock tak, jako bychom to mohli udělat my sami, jen to celé probíhá atomicky (nenarušitelně).

Ještě jeden rozdíl mezi upgradeable a read locky by si zasloužil zmínku. Zatímco upgradeable lock může bez problémů koexistovat s libovolným počtem read locků, pouze jeden upgradeable lock může být v jednu chvíli aktivní.

Teď si ukážeme využití upgradeable locku na příkladu. Dělá přesně to, co jsme si popsali ve dvou postupech výše.

```
while (true)
{
    // newNumber = Nějaké číslo z pomocné metody
    int newNumber = GetRandNum (100);
    rw.EnterUpgradeableReadLock();
    if (!items.Contains (newNumber))
    {
        rw.EnterWriteLock();
        items.Add (newNumber);
        rw.ExitWriteLock();
        Console.WriteLine ("Vlákno " + threadID + " přidalo číslo " +
newNumber);
```

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

3.3.1.1. Rekurzivní lock

Na závěr této kapitoly se jen krátce mrkneme na *rekurzivní locking*. Ve výchozí podobě je nějaké vložené nebo rekurzivní zamykání pomocí třídy *ReaderWriterLockSlim* zakázané. Takže tento kód se nezkompiluje:

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

Pokud se ale vytvoří instance *ReaderWriterLockSlim* takto, vše půjde bez problémů:

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

To, že musíme explicitně rekurzivní locking povolit zajišťuje, že k němu dojde tehdy, kdy to sami plánujeme:

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

Právě jsme probrali třídu *ReaderWriterLockSlim* a v zápětí se vrhneme na *thread pooling*.

3.4. Thread pooling (fond vláken)

3.4.1. Principy thread poolingu

Hned na úvod se zmíním o překladu pojmu „*thread pool*“. Většinou se setkáme s významem „*fond vláken*“, od toho odvozené sloveso „*thread pooling*“ by mohlo znamenat něco jako „*shromažďování vláken*“. Jak se totiž vzápětí dozvíte, „fond vláken“ je několik dohromady spojených vláken. V článku se ale raději budu držet anglického názvu - thread poolingu.

Teď už ale k využití thread poolingu. Pokud ve své aplikaci používáte hodně vláken, který většinu svého života tráví zablokovaný pomocí *Wait Handle*, můžete snížit použité systémové prostředky právě pomocí thread poooolingu, který spojí několik vláken do menšího počtu vláken.

K použití thread poolu musíte zaregistrovat Wait Handle společně s metodou, která se zavolá, když dojde k signalizaci Wait Handlu. Toho docílíme pomocí **ThreadPool.RegisterWaitForSingleObject**, jako v tomto příkladu:

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

    public static void Main()
    {
        ThreadPool.RegisterWaitForSingleObject(starter,  Go,  "ahoj",  -1,
true);
        Thread.Sleep(2000);
        Console.WriteLine("Signalizace pracovnímu vláknu...");
        starter.Set();
        Console.ReadLine();
    }

    public static void Go(object data, bool timedOut)
    {
        Console.WriteLine("Započato - " + data);
        // Provedení práce
    }
}
```

Výsledkem bude dvouvteřinová pauza, vypsání věty „Signalizace pracovnímu vláknu“ a „Započato – ahoj“.

Popíšeme si jednotlivé parametry metody. Podle MSDN má metoda tento předpis:

```
public static RegisteredWaitHandle RegisterWaitForSingleObject(WaitHandle
waitObject,
    WaitOrTimerCallback callBack,
    Object state,
    int millisecondsTimeOutInterval,
    bool executeOnlyOnce)
```

Teď následuje samotný popis parametrů:

- **waitObject** – Jakýkoliv *WaitHandle* kromě *Mutexu*.
- **callBack** – Přijímá delegát typu *Threading.WaitOrTimerCallback*, jež zastupuje metodu, která se má zavolat, když dojde k signalizaci *waitObjectu*.
- **state** – Objekt předaný metodě. Může to být cokoliv, následně se předá jako parametr delegované metodě (takže text „ahoj“ se předá jako parametr *data* metodě **Go**) vlastně stejně jako u **ParametrizedThreadStart** (vzpomínáte na první díl?).
- **milisecondsTimeOutInterval** - Časový údaj v milisekundách. Má stejnou funkci, jako všechny ostatní timeouty (hodnota -1 znamená žádný časový limit).
- **executeOnlyOnce** – Pokud je *true*, znamená to, že vlákno nebude čekat na *waitObject*, jestliže už došlo k zavolání delegátu *callBack*. Naopak *false* indikuje, že bude operace probíhat pořád dokola, dokud neodregistrujete *waitObject*.

Všechna vlákna ve fondu vláken pracují na pozadí (opakování: automaticky se zruší, jestliže přestanou existovat všechna vlákna běžící na popředí). Ovšem pokud bychom chtěli, aby se před ukončením aplikace nejdříve dokončila nějaká práce na vláknu ve fondu, zavolat metodu **Join** (jako bychom to udělali v běžné situaci - viz 3. kapitola) by nebylo řešením. Vlákna ve fondu totiž nikdy ve skutečnosti přirozeně neskončí! Místo toho se „recyklují“ (nefungují, ale nezmizí z paměti), přestanou existovat jedině tehdy, pokud přestane existovat i nadřízený proces. Takže abychom zjistili, jestli už vlákno v

thread poolu dokončilo svoji práci, museli bychom odeslat nějaký signál, třeba pomocí jiného Wait Handlu.

Můžeme použít thread pool i bez Wait Handlu, a to pomocí metody **QueueUserWorkItem**, které předáme delegát, jež se má okamžitě zavolat. Sice si tím znemožníte sdílení jednotlivých vláken pro několik prací, ale tento postup má i jednu výhodu. Thread pool si kontroluje celkový počet vláken (výchozí počet je 25) a automaticky vytváří frontu, pokud vznikne více úloh, než je vláken. V následujícím příkladu máme 100 úloh a 25 z nich probíhá v jeden okamžik. Primární vlákno pak pomocí metod **Wait** a **Pulse** čeká, dokud všechna pracovní vlákna nedodělají zadání.

```
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("Čekám na pracovní vlákna, až dodělají
práci...");
        lock (workerLocker)
        {
            while (runningWorkers > 0) Monitor.Wait(workerLocker);
        }
        Console.WriteLine("Hotovo!");
        Console.ReadLine();
    }

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

Na kódu by nemělo být co k nepochopení. Koneckonců, naprostou většinu jsme už dříve probrali.

Pokud bychom chtěli cílové metodě (v příkladu je to metoda **Go**) předat víc než jen jeden *object* parametr, máme několik možností. Můžeme použít anonymní metody. Kdyby metoda **Go** přijímala dva parametry typu *int*, mohli bychom delegát vytvořit takto:

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

Druhým způsob, jak se dostat do thread poolu, je přes asynchronní delegáty, na které se právě podíváme v následující kapitole.

3.5. Asynchronní delegáty

Ve druhé kapitole jsme si ukázali, jak předat data vláknu pomocí **ParametrizedThreadStart**. Někdy potřebujeme udělat opak – získat data od vlákna, jakmile dokončí svou práci. Asynchronní delegáty jsou pro toto nanejvýš vhodné, dovolují totiž předat libovolný počet argumentů v obou směrech. Navíc výjimky, které vzniknou na asynchronním delegátu jsou předány zpět volajícímu vláknu, to nám zajišťuje snadnější ošetření. Další výhoda byla zmíněna na konci předchozí kapitoly - díky asynchronním delegátům se můžeme dostat do thread poolu.

Nic není zadarmo, a tak i tady musíme zaplatit nějakou cenu. Tím je samotný asynchronní model, který je logicky o něco komplikovanější. Abyste lépe pochopili, o čem teď mluvím, ukážeme si jeden příklad vyřešený jak synchronně, tak asynchronně. Pokud bychom třeba chtěli porovnat obsah dvou stránek, jako první řešení by nás napadlo asi toto:

```
static void ComparePages()
{
    WebClient wc = new WebClient();
    string s1 = wc.DownloadString("http://www.programujte.com");
    string s2 = wc.DownloadString("http://chrasty.cz");
    // Na provedení chvíli počkáme...
    Console.WriteLine(s1 == s2 ? "Jsou stejné" : "Liší se");
}
```

Samozřejmě by bylo rychlejší stáhnout obě stránky naráz. Ale jak na to, když se další příkaz zavolá, až jakmile se dokončí ten předchozí? Ideální by bylo, kdyby to šlo následovně:

1. Zavoláme **DownloadString**.
2. Zatímco pracuje, budeme vykonávat jinou operaci, třeba stahování jiné stránky.
3. Řekneme si metodě **DownloadString** o výsledky.

Třetí krok je místo, kde jsou asynchronní delegáty užitečné. Volající metoda se setká s pracující metodou, společně vrátí nějaký výsledek a zároveň znovu vyhodí výjimky (pokud k nim v průběhu práce došlo), díky tomu je můžeme na tomto místě snadno ošetřit. Bez třetího bodu by se použití asynchronních delegátů nijak nelišilo od využití „obyčejného“ multithreadingu. Následující kód řeší stejný problém jako nahoře, jen asynchronně:

```
delegate string DownloadString(string uri);

static void ComparePages()
{
    // Vytvoříme instance delegátu DownloadString
    DownloadString download1 = new WebClient().DownloadString;
    DownloadString download2 = new WebClient().DownloadString;

    // Začneme stahovat
    IAsyncResult cookie1 = download1.BeginInvoke("http://programujte.com",
null, null);
    IAsyncResult cookie2 =
download2.BeginInvoke("http://www.programujte.com", null, null);

    // Získáme výsledky stahování, pokud je nutné, počkáme na dopočítání
    // Zde také dojde k vyhození výjimek
    string s1 = download1.EndInvoke(cookie1);
    string s2 = download2.EndInvoke(cookie2);
```

```
    Console.WriteLine(s1 == s2 ? "Jsou stejné" : "Liší se");
}
```

Na začátku deklarujeme a vytvoříme instance delegátu *DownloadString* pro metody, které chceme spustit asynchronně. V tomto případě potřebujeme dvě instance kvůli dvěma souborům, které chceme stáhnout.

Pak zavoláme **BeginInvoke**. To provede daný delegát a okamžitě vrátí kontrolu nad aplikací nám. Metodě **BeginInvoke** musíme předat tři parametry: cestu k souboru (stránce, která se má stáhnout), nepovinný callback (tedy metodu, která se má zavolat při volání delegátu) a jako třetí parametr můžeme zadat cokoliv – je typu *object*. Posledním dvěma se často nastavuje hodnota null, nejen v příkladu nahoře, protože většinou nejsou potřeba. Metoda **BeginInvoke** vrací objekt typu *IASynchResult*, jenž využijeme zároveň jako cookie (tedy nějaký vzorek dat) při volání **EndInvoke**. Objekt *IASynchResult* zároveň disponuje vlastností *IsCompleted*, díky které můžeme monitorovat průběh stahování.

Následně zavoláme zmiňovanou metodu **EndInvoke** s „cookie“ parametrem na delegáty, abychom získali výsledky. Pokud je to nutné, **EndInvoke** počká, dokud jeho metoda nedokončí práci, pak vrátí její výslednou hodnotu. Typ vrácených dat jsme nastavili v hlavičce delegátu (v našem případě tedy bude hodnota typu *string*, takže ji musíme uložit do proměnné typu *string*).

Na závěr se vyhodí případné výjimky, ke kterým došlo během asynchronního volání, a zde je můžeme jednoduše ošetřit (vím, tuto vymoženost zmiňuji už potřetí).

> Pokud metoda, kterou spouštíme asynchronně, nemá žádnou návratovou hodnotu, teoreticky nemusíme volat **EndInvoke**. Pak ale budeme muset případné výjimky ošetřit už na pracovním vláknu.

3.5.1. Asynchronní metody

Některé .NET typy poskytují asynchronní verze svých metod, typicky jejich název začíná na „Begin“ a „End“. Těmto metodám se říká asynchronní metody a mají podobné signatury jako asynchronní delegáty, ale používáme je pro řešení mnohem složitějšího problému – abychom mohli najednou provádět více operací, než máme k dispozici vláken. Například takový TCP socket server dokáže zpracovávat stovky všech možných požadavků a dotazů najednou, pokud použijeme metody **NetworkStream.BeginRead** a **NetworkStream.BeginWrite**, a přitom disponuje třeba jen několika vlákny v thread poolu.

Jestliže ale nejste v takto extrémní situaci, měli byste se použití asynchronních metod vyhnout hned z několika důvodů:

- Na rozdíl od asynchronních delegátů nemusí async. metody vždy běžet paralelně s tím, kdo je zavolal.

- Kód se za chvíli stane velmi složitým (jen si představte: synchronizace jednotlivých požadavků, zpracování, …), až se může stát, že výhoda async. metod za chvíli vymizí úplně.

Pokud vám jde jen o prosté paralelní spouštění metod, raději byste měli používat synchronní verze těchto metod pomocí asynchronních delegátů nebo použít *BackgroundWorker* anebo prostě vytvořit nové vlákno.

3.5.2. Asynchronní události

S asynchronními metodami se setkáme i zde, ve spojitosti s asynchronními událostmi. Z běžného programování znáte typickou dvojici – událost a metodu, která se zavolá, když dojde k události. To samé existuje i v asynchronních verzích, podle konvencí končí název takové metody na „Async“ a název události na „Completed“. Můžeme se s tím setkat například ve třídě *WebClient*, která definuje metodu **DownloadStringAsync**. Využití je následující: nejdříve zpracujeme událost **DownloadStringCompleted**, pak zavoláme **DownloadStringAsync**. Jakmile ta dokončí svou práci, zavolá se obsah zpracovatele události **DownloadStringCompleted**.

Máme k dispozici i několik událostí pro zpravování uživatele o průběhu nebo o zrušení akce, možná si pamatujete, že tyhle různé „Async“ a „Completed“ věci jsme viděli už v *BackgroundWorkeru*. Nebylo to nic jiného než asynchronní události.

To je k tomuto tématu vše. Ukázali jsme si několik asynchronních koutů multithreadingu a ve dvou dalších kapitolách nás čeká přehled timerů využitelných v threadingu a local storage.

3.6. Timery (časovače)

Nejjednodušším způsobem, jak zavolat nějakou metodu periodicky (po pravidelně se opakujících intervalech), je použití časovačů (timerů). V .NET frameworku je jich hned několik. Když už bereme vlákna, podíváme se nejdříve na ten ze *System.Threading.Timer*. Třída *Timer* je velice jednoduchá – obsahuje jen konstruktor a dvě metody (jaká úleva, není toho tolik k popisování ani k pamatování).

Definice této třídy vypadá následovně:

```
public sealed class Timer : MarshalByRefObject, IDisposable
{
    public Timer (TimerCallback tick, object state, prvniTick, interval);
    public bool Change (prvniTick, interval);    // změna intervalu
    public void Dispose();
}
    //PrvniTick - čas, jak dlouho má Timer
    // udělat první tick
    //interval - intervaly mezi dalšími ticky
    // použijte Timeout.Infinite, pokud chcete
    // jen jeden tick
```

V následujícím příkladu, jakmile zapnete program, se spustí odpočítávání 5 vteřin, pak se vypíše na obrazovku nápis „tik ťak“, který se bude opakovat každou vteřinu, dokud uživatel nestiskne Enter.

```
class Program
```

```
{
    static void Main()
    {
        using (new Timer(Tick, "tik ťak", 5000, 1000))
        {
            Console.ReadLine();
        }
    }

    static void Tick(object data)
    {
        // Spustí se na vlákně ve fondu vláken
        Console.WriteLine(data);
    }
}
```

Jak už jsem řekl, v .NETu existují i jiné timery. Teď se tedy podíváme na namespace *System.Timers*. Třída *Timer* z tohoto namespace obaluje tu ze *System.Threading*, přidává nějakou funkcionalitu navíc a pár změn:

- Je to komponenta, takže ho můžeme přetáhnout z Toolboxu a pracovat s ním ve Visual Studio Designeru.
- Má vlastnost *Interval* namísto metody **Change**.
- Má událost **Elapsed** namísto callback delegátu.
- Vlastnost *Enabled(bool)* pro spuštění a zastavení časovače (výchozí je *false*).
- Metody **Start** a **Stop** (pro případ, že by někomu nevyhovovala vlastnost *Enabled*).
- Vlastnost *AutoReset* pro indikaci, jestli se má časovač spouštět znovu (výchozí je *true*).

Většinu těchto změn ukazuje následující kód:

```
class SystemTimer
{
    static void Main()
    {
        var tmr = new Timer();  // Bez argumentů
        tmr.Interval = 500;
        tmr.Elapsed += tmr_Elapsed; // Událost místo callback delegátu
        tmr.Start();    // Spustí timer
        Console.ReadLine();
        tmr.Stop(); // Pozastaví timer
        Console.ReadLine();
        tmr.Start();    // Spustí timer (od předchozí hodnoty)
        Console.ReadLine();
        tmr.Dispose();  // Permanentně stopne timer
    }

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

Existuje ještě třetí timer, který se pro změnu nachází v namespace *System.Windows.Forms*. Radikálně se liší od timerů z *Threading* a *Timers*, protože nepoužívá thread pool, ale vždy vypaluje událost **Tick** na stejném vlákně, jako byl vytvořen. Pokud ho tedy vytvoříme na primárním vlákně, může přistupovat k ovládacímu prvku a měnit jej v závislosti na „tikání“ bez použití **Control.Invoke**. Další vlastnosti má společné s timerem ze *System.Timers*.

> WPF má ekvivalentní timer k tomu u WinForms, jen má jiný název – *DispatcherTimer*.

3.7. Local Storage

Dalším tématem (nesouvisejícím s timery), na které se podíváme, je Local Storage neboli „lokální úložiště“. S tímto pojmem jste se mohli setkat například u ASP.NET, Silverlightu a podobných technologií. Vypadá to tak, že každé vlákno dostane přidělené datové úložiště izolované od ostatních vláken. Je užitečné pro ukládání různých informací o zabezpečení, protokolů, ale dá se použít i k ukládání jiných dat. Kdybychom taková data předávali jako parametry metodám (pokud chceme, aby celé vlákno mělo k těmto datům přístup), bylo by to nepohodlné a měly by k nim přístup jen naše vlastní metody.

Dvě nejdůležitější metody jsou **Thread.GetData**, která umí číst data z Local Storage, a **ThreadSetData**, která naopak zapisuje. Obě metody potřebují ke své funkčnosti instanci třídy *LocalDataStoreSlot*, jež reprezentuje slot (představíme-li si Local Storage jako skříňku, pak je slot něco jako přihrádka - takových přihrádek můžeme mít tolik, kolik chceme). Konstruktor *LocalDataStoreSlot*u přijímá parametr typu *string*, který reprezentuje název slotu. Na více vláknech mohou mít sloty stejný název a přitom svoje data nesdílí. Je to kvůli tomu, že každé vlákno má svůj vlastní Local Storage. Příklad využití LS:

```
class Test
{
    // Stejný objekt LocalDataStoreSlot
    //můžeme použít napříč všemi vlákny
    LocalDataStoreSlot secSlot = Thread.GetNamedDataSlot("securityLevel");

    // Tato vlastnost bude mít na každém vlákně
    // jinou hodnotu
    int SecurityLevel
    {
        get
        {
            object data = Thread.GetData(secSlot);
            return data == null ? 0 : (int)data;
        }
        set
        {
            Thread.SetData(secSlot, value);
        }
    }

    // Další kód...
}
```

Metoda **Thread.FreeNamedDataSlot** zruší slot daného jména na všech vláknech, ale jen tehdy, pokud se už zadané sloty nepoužívají a byly sklizeny garbage collectorem.

To je k problematice Local Storage vše, stejně jako k třetí části knihy. Zbývá už jen poslední část, která se bude věnovat pokročilejším technikám, začneme klíčovým slovem *volatile* a *atomicitou*.

4. Část IV

4.1. Neblokující konstrukce

Na úplném začátku druhé části byla tabulka nejrůznějších metod synchronizace a na jejím konci byly konstrukce *volatile* a *Interlocked*. Jak už víte, synchronizace můžeme dosáhnout pomocí zamykání, ale to patří mezi blokující konstrukce – vlákno čeká, dokud není zámek otevřený. Naštěstí máme k dispozici neblokující konstrukce, které jsou vhodné pro velmi jednoduché a rychlé operace (tzv. atomické, vysvětleno níže), nedochází totiž k žádnému čekání ani blokování.

4.1.1. Atomicita a Interlocked

Slovo *atomicita* vám může připomínat jistě známější slovíčko *atom*; není to náhoda, obě jsou z řeckého slova *atomos*, tedy nedělitelný. Atomická operace se skládá jen z jedné nedělitelné operace (např. sečtení proměnných není atomická operace, protože se musí hodnoty načíst a pak teprve sečíst – sčítání je dělitelná operace). Atomickou operací je například u 32-bitových procesorů přiřazení čísla do proměnné typu *int*, která je 32-bitová.

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

    static void Test()
    {
        long myLocal;
        x = 3;              // Atomické
        z = 3;              // Neatomické (z je 64-bit)
        myLocal = z;        // Neatomické (z je 64-bit)
        y += x;             // Neatomické (čtení a zapisování)
        x++;                // Neatomické (čtení a zapisování)
    }
}
```

Práce s 64-bitovými čísly na 32-bitových procesorech není atomická operace, protože vyžaduje alokování dvou 32-bitových míst v paměti. Pokud nějaké vlákno *A* načítá 64-bitové číslo, zatímco vlákno *B* ho upravuje, může vlákno *A* dostat jakýsi mix obou hodnot (protože jedno 64-bitové číslo je složené vlastně ze dvou). Z tohoto odstavce je tedy jasné, proč práce s takovými čísly není atomická.

Atomické nejsou ani unární operátory (takové, které pracují s jednou proměnnou) typu *x++*. Nejdřív se musí aktuální hodnota „x“ načíst, pak přičíst jedničku a nakonec uložit novou hodnotu. Představte si takovouhle třídu:

```
class ThreadUnsafe
{
    static int x = 1000;
    static void Go() { for (int i = 0; i < 100; i++) x--; }
}
```

Možná byste čekali, že pokud metodu **Go** zavolá deset vláken najednou, proměnná *x* bude mít hodnotu 0 (cyklus proběhne 100x na deseti vláknech). To nám ale nikdo nezaručí, protože je možné,

že jedno vlákno přistoupí k proměnné, zatímco druhé bude získávat její hodnotu, snižovat ji a zapisovat zpátky.

Jedním ze způsobů, jak toto nebezpečí ošetřit, je obalit uvedenou (neatomickou) operaci do locku. Nikdy jsme si to neřekli, ale teď vám to možná došlo – locking vlastně udělá z obalené operace atomickou. Existuje ale druhý, výhodnější způsob. Ten provedeme pomocí třídy *Interlocked*, která je jednodušší a rychlejší, pokud ji použijeme pro jednoduché operace.

```
class Program
{
    static long sum;

    static void Main()
    {
        // Inkrementace/dekrementace:
        Interlocked.Increment(ref sum);      // to samé jako: sum++
        Interlocked.Decrement(ref sum);      // sum--

        // Přičtení/odečtení čísla:
        Interlocked.Add(ref sum, 3);         // sum += 3
        Interlocked.Add(ref sum, -2);        // sum -= 2

        // Přečtení hodnoty 64-bit čísla
        Console.WriteLine(Interlocked.Read(ref sum));   // sum == 1

        // Přečte hodnotu a pak zapíše novou
        // Následující řádek napíše "1" a pak změní hodnotu
        // sum na 10
        Console.WriteLine(Interlocked.Exchange(ref sum, 10));     // sum ==
10

        // Změní hodnotu proměnné, ale jen pokud se
        // momentálně rovná zadané hodnotě (10)
        Interlocked.CompareExchange(ref sum, 123, 10);  // sum == 123

        // Finální hodnota proměnné sum:
        Console.WriteLine(Interlocked.Read(ref sum));

        Console.ReadKey();
    }
}
```

Používání třídy *Interlocked* je výhodné, protože obsahuje už předpřipravené atomické metody pro hojně používané operace. Zároveň nemůže dojít k jejímu zablokování, takže nemusíme nést následky přerušení práce.

4.1.2. Memory barriers a volatilita

Vezměme si tento kód:

```
class Unsafe
{
    static bool konec, boolean;

    static void Main()
    {
        new Thread(Wait).Start();
        Thread.Sleep(1000);
        boolean = true;
        konec = true;
        Console.WriteLine("Něco se děje...");
    }

    static void Wait()
    {
        while (!konec) ;
        Console.WriteLine("A je klid, " + boolean);
        Console.ReadKey();
    }
}
```

Prohlédněte si jej. Nedělá nic komplikovaného: máme metodu, která se zavolá a je uzavřená v cyklu. Po jedné vteřině ji z toho cyklu osvobodíme nastavením proměnné *konec* na *true* a metoda pak vypíše: „A je klid“ společně s hodnotou proměnné *boolean*.

Teď si položme otázky: je možné, aby metoda **Wait** byla pořád uzavřená ve *while* cyklu i po tom, co se proměnná *konec* nastaví na *true*? A je vůbec možné, aby metoda **Wait** napsala: „A je klid, False“?

Vypadá to nepravděpodobně, že? Ale odpověď na obě otázky je ano. Na víceprocesorových strojích, jakmile se každé vlákno přidělí na jiný procesor, se může stát, že se obě proměnné *konec* a *boolean* uloží do cache (vyrovnávací paměti), aby se k nim umožnil rychlejší přístup. Hrozí ale prodleva mezi tím, než se zapíší zpátky do paměti, a nemusí se nutně zapsat ve stejném pořadí, jako se uložily.

Toto riziko můžeme obejít použitím statických metod **Thread.VolatileRead** a **Thread.VolatileWrite** při práci s proměnnými. **VolatileRead** vlastně znamená „přečti poslední hodnotu“ a **VolatileWrite** zase „zapiš okamžitě do paměti“. Stejného výsledku dosáhneme i elegantněji - deklarováním proměnné jako *volatile* (v překladu „nestálý“, stejně jako paměť RAM):

```
volatile static bool konec, boolean;
```

Pokud proměnnou deklarujeme takto, říkáme tím vlastně: „nekešuj tuhle proměnnou“.

Stejného výsledku bychom dosáhli i použitím prostého locku. Fungovalo by to, protože vedlejším efektem zamykání je vytvoření tzv. „*memory barrier*“ – máme jistotu, že při vstupu do locku bude mít proměnná svojí nejaktuálnější hodnotu a před opuštěním locku se poslední hodnota zapíše do paměti.

Použít tento postup by bylo nutné v případě, že bychom potřebovali přistupovat k proměnným *konec* a *boolean* atomicky, například takhle:

```
lock (locker) { if (konec) boolean = true; }
```

Volatilita se týká jen primitivních typů, jiné typy se necachují a nemůžou být ani deklarovány s klíčovým slovem *volatile*.

V další kapitole nás čekají metody **Wait** a **Pulse** – poslední dvě synchronizační konstrukce, které jsme ještě neprobrali.

4.2. Wait a Pulse

V kapitole 2.5. jsme probrali třídy odvozené od abstraktní třídy *WaitHandle* – jednoduché signalizační konstrukce, kde se vlákno zablokuje, než dostane impulz od jiného vlákna.

Mnohem mocnější signalizační konstrukce nám poskytuje třída *Monitor* pomocí dvou statických metod. Jak už možná tušíte, jsou to metody **Wait** a **Pulse**. Ve zkratce to vypadá tak, že si celou signalizační logiku můžeme napsat sami, můžeme okopírovat funkcionalitu tříd *AutoResetEvent*, *Semaphore*, … a ještě si další věci přidat.

Největším problémem **Wait** a **Pulse** je jejich chabá dokumentace, částečně určitě způsobený tím, že využití nenajdou tak moc často. Abychom to neměli příliš jednoduché, tyto metody dokážou v kódu způsobit hotový chaos, pokud nevíme přesně, co děláme. Naštěstí existuje doporučený model použití, a pokud se jím řídíme, žádné chyby nám nehrozí. Na tento model se podíváme za chvíli.

4.2.1. Úvod k Wait a Pulse

Účelem metod **Wait** a **Pulse** je poskytnutí jednoduchého signalizačního mechanismu: **Wait** zablokuje vlákno, dokud nedostane z jiného vlákna signál od metody **Pulse**.

Je logické, že **Wait** musí být zavoláno před **Pulse** (jinak by nebylo komu co signalizovat). Pokud ale nějakým nedopatřením přesto dojde k zavolání **Pulse** dříve, vůbec nic se nestane. Tady si můžete všimnout rozdílu oproti *AutoResetEventu*, u kterého se efekt odloží, pokud dojde k zavolání metody **Set** před **WaitOne**.

Když chceme v naší aplikaci použít **Wait**/**Pulse**, musíme definovat synchronizační objekt. Princip je jednoduchý – platí, že pokud obě vlákna používají stejný synchronizační objekt, můžou si mezi sebou posílat signály pomocí **Wait** a **Pulse**. Druhou důležitou věcí je, že synchronizační objekt musí být vždy uzamknut v locku, než ho použijeme při volání **Wait** nebo **Pulse**.

Co jsme si teď pověděli si ukážeme na příkladu:

```
class Test
{
    // Synchronizační objekt 'x'
    // Jako sync. obj. poslouží cokoliv
    // referenčního typu
    object x = new object();
}
```

Tento kód zablokuje vlákno „*A*“:

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

A při zavolání tohoto kódu z vlákna „*B*“ dojde k odblokování vlákna „*A*“:

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

4.2.1.1. Jak to funguje?

Dokud vlákno čeká, metoda **Monitor.Wait** dočasně uvolňuje zámek kolem objektu „*x*“, aby ho mohlo jiné vlákno (to, které volá **Monitor.Pulse**) zase uzamknout. Celý proces můžeme vlastně napsat takto:

```
Monitor.Exit (x);              // Odemknutí zámku
// čekání na Pulse…
Monitor.Enter (x);             // Uzamknutí zámku
```

Proto se může **Wait** zablokovat ve skutečnosti dvakrát: poprvé, když čeká na **Pulse** a pak když se čeká na znovu-uzamknutí zámku. To také znamená, že **Pulse** neodblokuje čekající vlákno úplně, jen jakmile vlákno, které zavolalo **Pulse**, opustí kód v bloku lock může čekající vlákno skutečně začít pracovat.

Principu výše se říká *lock toggling*. Lock toggling nijak nezávisí na úrovni vnoření jednotlivých locků. Pokud třeba zavoláme **Wait** uvnitř dvou vložených locků:

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

Znovu si můžeme představit přepsání na **Monitor.Exit** a **Monitor.Enter**:

```
Monitor.Exit (x); Monitor.Exit (x);     // 2 locky = 2 Exit
// čekání na Pulse…
Monitor.Enter (x); Monitor.Enter (x);
```

4.2.1.2. Proč musíme lockovat?

Abych upřesnil nadpis: proč jsou vůbec metody **Wait** a **Pulse** vytvořeny tak, aby fungovaly jen uvnitř locku? Prvním důvodem je samozřejmě to, aby nehrozilo narušení thread-safety. Řekněme, že chceme zavolat **Wait** jen pokud je proměnná *dostupny* nastavená na *false*.

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

Několik vláken najednou může spustit tento kód, ale žádné se nemůže přerušit mezi kontrolováním hodnoty *dostupny* a voláním **Monitor.Wait** (ano, je to atomické). Odpovídající operace s **Pulse** by vypadala takto:

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

4.2.1.3. Nastavení timeoutu

Timeout můžeme nastavit při volání **Wait** buďto klasicky v milisekundách nebo jako *TimeSpan*. Metoda **Wait** pak vrátí *false*, pokud vypršel timeout dřív, než se stihla zavolat **Pulse**. Tento časový limit ovlivní pouze čekací fázi, pokud ale vyprší, čekání bude probíhat stále! Příklad:

```
lock (x)
{
    if (!Monitor.Wait (x, TimeSpan.FromSeconds (10)))
        Console.WriteLine ("Čas vypršel!");
    Console.WriteLine ("Ale zámek na x je pořád.");
}
```

4.2.2. Vlastnosti a nevýhody Pulse

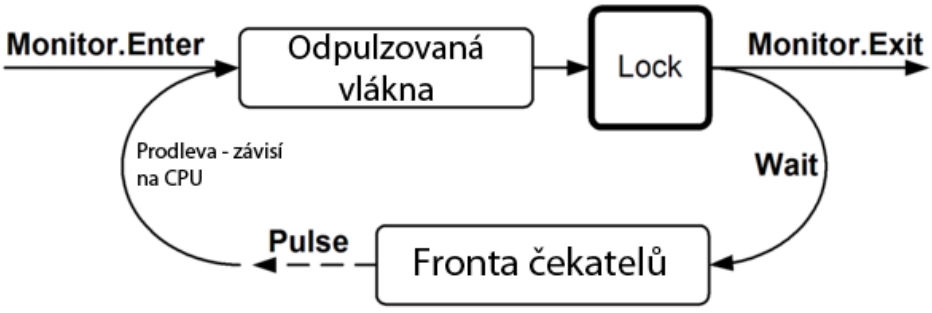
Důležitou vlastností metody **Pulse** je to, že je volána asynchronně, tedy se nemůže nijak zablokovat. Pokud nějaké jiné vlákno čeká na impulz, dostane ho. Pokud žádné vlákno nečeká, efekt je ignorován.

Pulse disponuje jen jednosměrnou komunikací – vlákno posílající **Pulse** odešle signál čekajícímu vláknu. Nic víc. **Pulse** nevrací žádnou hodnotu indikující, jestli došel impulz k cíli, nebo jestli byl ignorován. Navíc, i když impulz skutečně k cíli dorazí, nemáme žádnou záruku, že čekající vlákno hned znovu nastoupí do služby. Může nastat dlouhá prodleva, než se procesor k tomuto vláknu vůbec dostane. Kvůli všem těmto komplikacím (a dost možná i nedokonalostem) by bylo těžké zjistit, kdy bylo vlákno odblokováno, kdy začalo pracovat, … jinak než prostřednictvím nějakých pomocných proměnných, které si sami definujeme. Proto nikdy nespoléhejte jen na časové odezvy při použití **Wait** a **Pulse**, mohli byste narazit.

4.2.3. Fronty čekatelů a PulseAll

Zavolat **Wait** na jeden objekt může najednou víc než jen jedno vlákno, tím se vytvoří tzv. „fronta čekatelů“ (*waiting queue*).

Každé zavolání **Pulse** pak uvolní vlákno na začátku fronty (je to tedy typ *FIFO – First In, First Out*), které se přesune do fronty odpulzovaných vláken a tam čeká na znovuobdržení zámku, stejně jako na obrázku:



Ovšem pozor. Pořadí, které vlákna získají tímto seřazením do fronty většinou nehraje roli, protože v aplikacích využívajících **Wait** a **Pulse** se nejedná tak úplně o frontu, jako spíše o vanu plnou čekajících vláken, ze kterých pak zavolání Pulse jedno vybere a propustí.

Třída *Monitor* poskytuje také metodu **PulseAll**, která, jak už název napovídá, propustí všechna čekající vlákna ve frontě (tedy vaně, chcete-li) najednou. Propuštěná vlákna nezačnou pracovat

nastejno, protože všechna čekají na uzamknutí stejného objektu, tedy se znovu vytvoří fronta. Pokud bychom se podívali na obrázek, tak metoda **PulseAll** přesune všechna vlákna z „fronty čekatelů“ do fronty „odpulzovaných vláken“, kde čekají na získání zámku.

4.2.4. Jak použít Pulse a Wait

Definujme si dvě pravidla:

- Jediná dostupná synchronizační konstrukce je lock (tedy **Monitor.Enter** a **Monitor.Exit**)
- Nejsou žádná omezení co se spinningu týče

S těmito pravidly na vědomí si ukažme následující příklad: pracovní vlákno, které čeká, dokud nedostane signál od hlavního vlákna.

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

    void Work()
    {
        Console.Write("Čekám... ");
        lock (locker)
        {
            while (!go)
            {
                // Uvolníme zámek, takže hlavní vlákno bude moct
                // změnit hodnotu 'go'
                Monitor.Exit(locker);
                // Znovu uzamkneme zámek, abychom mohli znovu
                // otestovat hodnotu 'go' na začátku cyklu
                Monitor.Enter(locker);
            }
        }
        Console.WriteLine("Dostal jsem signál!");
    }

    void Notify() // Voláno z hlavního vlákna
    {
        lock (locker)
        {
            Console.Write("Signalizuji... ");
            go = true;
        }
    }
}
```

Abychom mohli kód spustit, potřebujeme ještě metodu Main:

```
static void Main()
{
    SimpleWaitPulse test = new SimpleWaitPulse();

    // Spustíme metodu Work na samostatném vlákně
    new Thread(test.Work).Start();  // "Čekám..."

    // Za vteřinu pozastavíme, pak odešleme signál:
```

```
    Thread.Sleep(1000);
    test.Notify();  // "Signalizuji... Dostal jsem signál!"
}
```

V metodě **Work** dochází ke spinningu - úplně zbytečně spotřebováváme procesorový čas tím, že neustále opakujeme obsah while cyklu, dokud nemá proměnná go hodnotu *true*. V tomto cyklu musíme odemykat a znovu uzamykat zámek, aby metoda **Notify** mohla uzavřít svůj lock a změnit hodnotu *go*. *go* je sdílená uvnitř celé třídy, proto k ní musíme přistupovat uvnitř zámku, abychom měli jistotu, že se její hodnota nezmění někde „mezi“ (nemůžeme použít *volatile*, jak jsme si řekli na začátku téhle kapitoly).

Zkuste kód uvedený výše zkompilovat, dostanete text: *„Čekám… (vteřinová pauza) Signalizuji… Dostal jsem signál!“*.

Teď pojďme náš kód upravit tak, aby namísto **Monitor.Exit** a **Enter** používal metody **Wait** a **Pulse** (výstup do konzole je vynechán naschvál, ať je to stručné):

```
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);
        }
    }
}
```

Kód se chová stejně jako předtím, jen s jedním důležitým rozdílem – nedochází k žádnému spinningu. Metoda **Wait** dělá to samé co **Monitor.Exit** a **Monitor.Enter**, ale s žádným mezikrokem – když se zámek otevře, čekáme na zavolání **Pulse**. O to se postará metoda **Notify**, jakmile nastaví *go* na *true*. To je vše.

4.2.5. Model použití Pulse a Wait

Na začátku dnešního dílu jsme si řekli, že použití **Pulse** a **Wait** se může rychle zvrhnout ve velmi složité bludiště. V této kapitole ho prozkoumáme a ukážeme si ho na kousku pseudo-kódu. Tento vzor následoval už příklad v předchozí kapitole a silně doporučuji se ho držet i ve vašich aplikacích.

V kódu v předchozí kapitole jsme měli jen jednu proměnnou, které se týkalo zamykání (*go*). V jiné situaci bychom takových mohli potřebovat víc, postup je ale stejný jako s jednou proměnnou. Nejdřív si vzor ukážeme s použitím spinningu:

```
class X
{
    //Blokované proměnné: jeden nebo více objektů zahrnutých v blokování,
např:
    //bool go; bool ready; int semaphoreCount;...

    // locker chrání proměnné uvedené výše
    object locker = new object();

    void MetodaNaSpinning()
    {
        //... když chci vlákno zablokovat na základě proměnných
        lock (locker)
        {
            while (!) // Seznam podmínek podle gusta
            {
                // Dáme ostatním vláknům šanci k upravení proměnných
                Monitor.Exit(locker);
                Monitor.Enter(locker);
            }
        }
    }

    void MetodaNaUpravu()
    {
        //... když chci upravit proměnné
        lock (locker)
        {
            //tady pracujte s proměnnými
        }
    }
}
```

Pokud opět spinning nahradíme metodami **Pulse** a **Wait**:

- Exit a Enter nahradíme metodou **Wait**
- Když upravíme hodnoty proměnných, zavoláme **Pulse** těsně před uvolněním zámku

Pseudo-kód bude vypadat takto:

```
class X
{
    //< Blokované proměnné ... >
    object locker = new object();

    void WaitMetoda()
    {
        //...
        //... když chci vlákno zablokovat na základě proměnných
        lock (locker)
        {
            while (!) // Seznam podmínek podle gusta
            {
                Monitor.Wait(locker);
            }
        }
    }


    void PulseMetoda()
```

```
    {
        //... když chci upravit proměnné
        lock (locker)
        {
            //tady pracujte s proměnnými

            Monitor.Pulse(locker);
        }
    }
}
```

Ukázali jsme si bezpečný vzor použití **Wait** a **Pulse**, jeho hlavní výhody by se daly shrnout do několika bodů:

- Blokovací podmínky jsou v podobě námi definovaných proměnných (které jsou logicky schopné fungovat i bez **Wait** a **Pulse** i při spinningu).
- **Wait** je volána vždy uvnitř while cyklu, kde kontroluje blokovací podmínky (a cyklus sám je uvnitř locku).
- Jeden jediný synchronizační objekt (v příkladu jsme použili locker) je použit pro všechna **Wait** a **Pulse** a ochraňuje tak všechny blokovací podmínky najednou.
- Locky jsou tam jen kvůli nutnosti, ale vždy můžeme bez problémů zámek opustit.

Co je možná nejdůležitější - pokud budeme následovat tento vzor, zavolání **Pulse** nenutí čekatele, aby pokračoval v práci. Místo toho mu jen oznámí, že se „něco stalo“ a že by měl znovu zkontrolovat platnost blokovacích podmínek. Čekatel pak sám usoudí (dalším průchodem cyklu), jestli by měl znovu čekat, nebo cyklus opustit a pokračovat v práci. Výhoda je, že můžeme používat složité blokovací podmínky bez nějaké složité synchronizace.

Další výhodou je odolnost proti špatně zavolanému **Pulse**. To se stane, pokud se **Pulse** zavolá před **Wait**. Ale protože v tomto vzoru zavolání **Pulse** znamená „zkontroluj blokovací podmínky“ a ne „okamžitě pokračuj v práci“, může být příliš brzké zavolání **Pulse** ignorováno, protože než se zavolá samotné **Wait**, vždy se zkontroluje i blokovací podmínka.

Poslední poznámku mám k synchronizačnímu objektu. Díky tomu, že je jen jeden, můžeme přistupovat k proměnným atomicky. Pokud bychom měli pro lock, **Pulse** a **Wait** jiný synchronizační objekt, mohlo by docházet k deadlockům. Doporučuje se také deklarovat synchronizační objekt a jeho proměnné jen v rozsahu, kde je budeme skutečně potřebovat (tohoto omezení dosáhneme např. pomocí modifikátoru *private*).

4.2.6. Fronta producent/spotřebitel

Jednoduchou aplikací, která **Wait/Pulse** využije, je fronta producent/spotřebitel (tu jsme si ukázali v kapitole 2.5.2.2.). Nějaký kód, označovaný jako producent, přidává do fronty zadání (typicky na hlavním vlákně), zatímco jeden nebo více spotřebitelů odebírají zadání jedno po druhém a plní je.

V našem příkladu použijeme na reprezentaci úkolu typ *string*, fronta tedy bude vypadat takto:

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

Protože budeme k frontě přistupovat z několika vláken, musíme všechen kód, který z ní čte nebo do ní zapisuje, obalit do locku. Takhle budeme přidávat úkoly do fronty:

```
lock (locker)
{
    taskQ.Enqueue("můj úkol");
    Monitor.PulseAll(locker);
}
```

Upravujeme potenciální blokovací podmínku, takže musíme zavolat **Pulse**. **PulseAll** namísto něj voláme, protože můžeme mít více spotřebitelů, tedy více čekajících vláken.

Je žádoucí, aby se spotřebitelé zablokovali, pokud nemají zrovna co dělat (pokud je fronta prázdná), aby zbytečně nezatěžovali počítač. Tento kód dělá přesně to, co chceme – kontroluje počet prvků ve frontě.

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

Dalším krokem je, aby spotřebitel mohl odebrat úkol z fronty a splnit ho:

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

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

Právě uvedený postup ale není thread-safe. Odstranění z fronty se totiž zakládá na staré informaci z uzamknutí locku. Představte si, co by se stalo, kdybychom spustili dvě spotřebitelská vlákna najednou s jedním předmětem umístěným ve frontě. Mohlo by se stát, že ani jedno by se uvnitř while cyklu nezablokovalo, protože by obě ve stejný moment viděla právě ten jeden předmět. Hned potom by se obě vlákna pokusila úkol z fronty odstranit a právě v této chvíli by došlo k chybě. Abychom tomuto předešli, budeme udržovat lock zamčený o chvilku déle:

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

Po odstranění úkolu z fronty už nemusíme volat **Pulse**, protože žádný spotřebitel se neodblokuje jen kvůli tomu, že ubyly úkoly.

Jakmile je úkol z fronty pryč, není nutné pořád udržovat lock. Tím, že ho teď odemkneme, umožníme spotřebiteli provést časově náročný úkol, aniž by při tom blokoval ostatní vlákna.

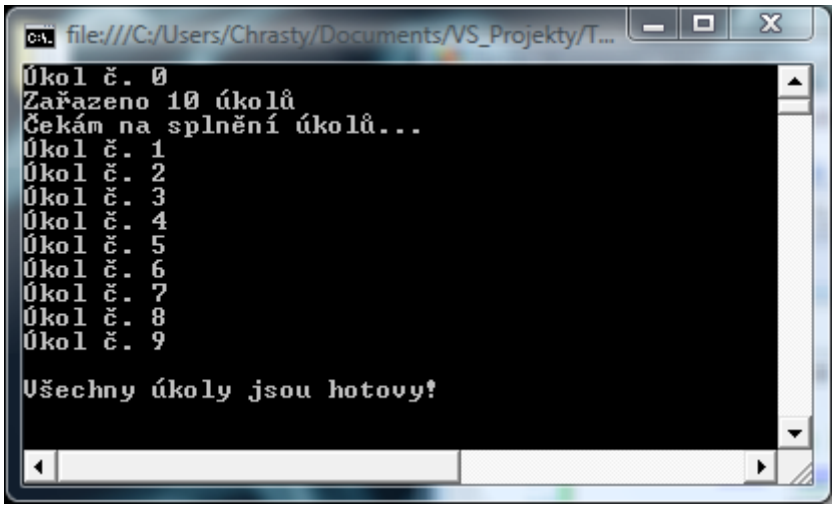
Ukážeme si kompletní program, který jsme právě skládali dohromady. Stejně jako u verze, kde jsme použili *AutoResetEvent*, použijeme *null* úkol, abychom oznámili spotřebiteli, že je konec. Protože aplikace podporuje více než jednoho spotřebitele, musíme do fronty zařadit odpovídající počet *null* úkolů, aby se práce ukončila u všech.

```
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];

        // Pro každého spotřebitele vytvoříme vlákno
        for (int i = 0; i < workerCount; i++)
            (workers[i] = new Thread(Consume)).Start();
    }

    public void Dispose()
    {
        // Do fronty zařadíme tolik null úkolů, kolik je vláken
        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;   // konec
            Console.WriteLine(task);
            Thread.Sleep(1000); // Simulace časově náročného úkolu
        }
    }
}
```

A zde je metoda **Main**, která vše nastartuje, vytvoří dvě spotřebitelská vlákna a deset úkolů, které si mezi sebou rozdělí:

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

```
            q.EnqueueTask("Úkol č. " + i);

        Console.WriteLine("Zařazeno 10 úkolů");
        Console.WriteLine("Čekám na splnění úkolů...");
    }

    // Použili jsme using, takže po skončení práce se zavolá Dispose()
    // na spotřebitelská vlákna
    Console.WriteLine("\r\nVšechny úkoly jsou hotovy!");
}
```



4.2.7. Zátěž Pulse

Řekneme si, jestli by se nějak dal zrychlit proces pulzování. Nejdříve si zopakujme kód z metody **EnqueueTask** výše:

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

Teoreticky bychom mohli volat **PulseAll** jen tehdy, je-li vůbec možné nějaké vlákno odblokovat:

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

Ale pozor, moc bychom neušetřili (skoro vůbec nic), vzhledem k tomu, že už pulzování samotné je záležitost necelé mikrosekundy. Naopak bychom si mohli přitížit, podívejte se totiž na následující kód (a najděte rozdíl):

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

Přesně to je snad ten nejhorší typ bugů. Kompilátor chybu nenahlásí, vše klidně až do poslední chvíle funguje, jak má a kdo by pak v tom všem kódu hledal chybějící rovnítko?

4.2.8. Pulse nebo PulseAll?

Tady přichází na řadu další potencionální zúspornění kódu. Po zařazení úkolu do fronty bychom mohli zavolat namísto **PulseAll** „obyčejné“ Pulse a nic by se nestalo.

Zopakujme si rozdíly: když zavoláte **Pulse**, může se probudit k životu maximálně jedno vlákno. Pokud použijeme **PulseAll**, probudí se všechna. Když přidáváme do fronty jen jeden úkol, pouze jeden spotřebitel ho může zpracovat, takže nám stačí probudit jednoho pracovníka pomocí **Pulse**.

V příkladu níže máme jen dvě spotřebitelská vlákna, takže výkonnostní rozdíl mezi **Pulse** a **PulseAll** bude minimální. Pokud bychom ale měli takových vláken deset, bylo by o trošku výkonnostně výhodnější použít **Pulse** (i když ho volat desetkrát) než **PulseAll**:

```
lock (locker)
{
    taskQ.Enqueue("úkol 1");
    taskQ.Enqueue("úkol 2");
    Monitor.Pulse(locker);
    Monitor.Pulse(locker);
}
```

Cenou za rychlejší volání je zaseknutí pracovního vlákna. To je zase jeden z těžko odhalitelných bugů, projeví se až když je spotřebitel ve stavu *Waiting*. Dalo by se říct, že pokud jste na pochybách, jestli by k takovému bugu nemohlo dojít i ve vaší aplikaci, tak používejte **PulseAll**. Ztrátu výkonu nejspíš ani nezaznamenáte.

4.2.9. Použití timeoutu při Wait

Může se stát, že nebude vhodné zavolat **Pulse** hned, jakmile se splní blokovací podmínka. Příkladem takové situace je metoda, která získává informace pravidelným dotazováním se databáze. Pokud nám nevadí menší prodlevy, je řešení jednoduché – metodě **Wait** přidáme parametr timeout.

```
lock (locker)
{
    while (!podminka)
    Monitor.Wait(locker, timeout);
}
```

Tento postup přikáže znovu zkontrolovat platnost podmínky podle zadaného času a pak ještě jednou po obdržení pulzu. Čím je podmínka jednodušší, tím může být timeout menší při zachování účinnosti.

Timeouty se hodí i pokud by se mohlo stát, že se nebude moci zavolat **Pulse** (ať už kvůli bugu, je-li synchronizace složitá, nebo z jiného důvodu). Podmínka se pak zkontroluje i bez pulzování a aplikace může pokračovat.

4.2.10. Lock race a co s ním

Chceme-li zasignalizovat pracovní vlákno pětkrát za sebou, mohli bychom použít následující kód:

```
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("Co je?");
        }
    }
}
```

Očekávaný výstup by byla pětkrát otázka: „Co je?“. Ale to se nestane, místo toho se napíše jen jednou!

Kód je totiž vadný. Cyklus for v metodě **Main** může projet svých pět iterací, když ještě pracovní vlákno nedrží zámek nad objektem *locker*. Možná se projde cyklem ještě dřív, než se vůbec stihne pracovní vlákno nastartovat! Příklad v kapitole Fronta producent/spotřebitel netrpěl tímhle neduhem, protože pokud se hlavní vlákno dostalo v práci před pracovní vlákno, každý požadavek se jednoduše zařadil do fronty. Ale v tomto případě musíme hlavní vlákno zablokovat v každé iteraci, pokud je pracovní vlákno pořád zaměstnáno svým předešlým úkolem.

Jednoduchým řešením je, aby hlavní vlákno po každé iteraci počkalo, dokud nebude proměnná *go* nastavena na *false* pracovním vláknem. Kvůli této změně musíme volat i **Pulse**.

```
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);
            }
            Console.WriteLine("Co je?");
        }
    }
}
```

Důležitou vlastností tohoto programu je to, že pracovní vlákno uvolní zámek, než se vrhne na svůj (možná dlouhý) úkol (tady je úkolem zavolání **Console.WriteLine**).

V našem příkladu jen jedno vlákno (hlavní) signalizuje pracovnímu vláknu, aby pracovalo. Pokud bychom ale měli taková vlákna dvě a obě by volala kód podobný tomu v metodě **Main**, mohl by se následující řádek kódu zavolat dvakrát za sebou.

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

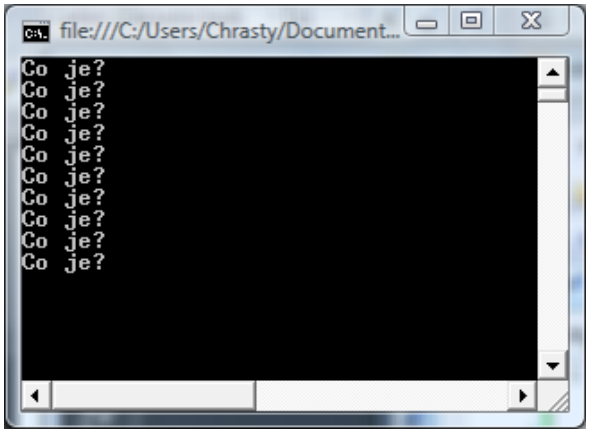
Výsledkem by bylo, že by druhá signalizace neměla žádný účinek, pokud by pracovní vlákno mezitím nestihlo svojí úlohu dokončit. Tohle můžeme ošetřit dvojicí proměnných – *ready* a *go*. *ready* indikuje, že je pracovní vlákno připraveno přijmout další zadání, zatímco *go* znamená příkaz k práci stejně jako předtím. Je to analogické k dřívějšímu příkladu se dvěma *AutoResetEventy*, jen flexibilnější. Takhle tedy bude vypadat kód:

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

    public void NotifyWhenReady()
    {
        lock (locker)
        {
            // Čekat, pokud má pracovní vlákno práci
            while (!ready) Monitor.Wait(locker);
            ready = false;
            go = true;
            Monitor.PulseAll(locker);
        }
    }
```

```
    public void AcknowledgedWait()
    {
        // Řekneme, že jsme připraveni na další úkol
        lock (locker) { ready = true; Monitor.Pulse(locker); }

        lock (locker)
        {
            while (!go) Monitor.Wait(locker);          // Počkáme na "go"
signál
            go = false; Monitor.PulseAll(locker);   // Přenastavíme "go",
zapulzujeme
        }

        Console.WriteLine("Co je?");   // Splníme úkol
    }
}
```

Pro demonstraci použijeme dvě vlákna, každé z nich pošle pětkrát signál pracovnímu vláknu. Mezitím hlavní vlákno čeká na deset zpráv.

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

    static void Main()
    {
        new Thread(Notify5).Start();      // Spustíme
        new Thread(Notify5).Start();      // dvě vlákna
        Wait10();   // ... a jednoho čekatele.
        Console.ReadKey();
    }

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

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



V metodě **NotifyWhenReady** je proměnná *ready* nastavena na *false* před opuštěním zámku a na tom staví celý příklad! Zabraňuje to totiž dvěma vláknům poslat signál za sebou bez zkontrolování hodnoty ready. Pro zjednodušení ve stejném zámku nastavíme i proměnnou *go* a zavoláme **PulseAll**, i když bychom mohli tyto dvě věci přesunout do jiného locku a nic by se nestalo.

4.2.11. Simulace třídy WaitHandle

V předchozím kódu jste si mohli všimnout, že oba while cykly mají takovouhle strukturu:

```
lock (locker)
{
    // Čekat, pokud má pracovní vlákno práci
    while (!promenna) Monitor.Wait(locker);
    promenna = false;
    ...
```

Přičemž *promenna* je nastavena na *true* v jiném vlákně. Tato struktura kódu kopíruje *AutoResetEvent*, třídu odvozenou od *WaitHandle*. Pokud bychom přehlédli kód "*promenna = false;*", dostali bychom *ManualResetEvent*. A pokud bychom místo *boolean* proměnné použili *int*, dostali bychom *Semaphore*. Ve skutečnosti jediná třída, jejíž funkci nemůžeme pomocí **Wait** a **Pulse** okopírovat, je *Mutex*, která dělá to samé co lock.

Simulovat statické metody, které fungují napříč několika *Wait Handly* je většinou jednoduché. Ekvivalentem k zavolání **WaitAll** na několik *EventWaitHandlů* není ve skutečnosti nic jiného než blokovací podmínka, která použije proměnné namísto *Wait Handlů*:

```
lock (locker)
{
    while (!promenna1 && !promenna2 && !promenna3...) Monitor.Wait
(locker);
```

Tento kód může být někdy užitečný, protože **WaitAll** je často nepoužitelný kvůli jeho COM historii. Simulování **WaitAny** je stejně jednoduché, jen nahraďte && operátorem ||.

SignalAndWait je už zapeklitější. Vzpomeňte si, že tato metoda signalizuje jeden handle, zatímco čeká na jiný, v jedné atomické operaci. Předpokládejme, že chceme poslat signál proměnné *A*, zatímco čekáme na proměnnou *B*. Museli bychom každou proměnnou rozdělit do dvou, ve výsledku by kód mohl vypadat následovně:

```
lock (locker)
{
    Acast1 = true;
    Monitor.Pulse (locker);
    while (!Bcast1) Monitor.Wait (locker);

    Acast2 = true;
    Monitor.Pulse (locker);
    while (!Bcast2) Monitor.Wait (locker);
}
```

Ve stejnou chvíli se na obrazovku vypíše dvakrát „Hej!“ – vlákna se úspěšně setkala.

4.2.12. Wait a Pulse vs. WaitHandle

Protože jsou **Wait** a **Pulse** rozhodně nejflexibilnější ze všech synchronizačních konstrukcí, najdou využití téměř v každé situaci. Přesto mají *WaitHandle* konstrukce dvě výhody:

- Mají schopnost pracovat napříč procesy
- Jsou jednodušší na pochopení a těžší na „rozbití“

Co se výkonu týče, pokud s **Wait** a **Pulse** postupujeme podle vzoru, který je:

```
lock (locker)
    while ( blokovací podmínka ) Monitor.Wait (locker);
```

Je to kvůli zamykání a znovuodemykání zámku, **WaitHandle.WaitOne** je prostě o trošičku rychlejší.

> Pokud vezmeme v úvahu různé procesory, operační systémy, verze CLR a samotnou logiku aplikací, dostaneme se k rozdílu maximálně pár mikrosekund mezi **Wait**/**Pulse** a *WaitHandle*. Tento rozdíl téměř určitě žádnou škodu nenadělá, proto si můžete vybrat vhodný nástroj podle aktuální situace.

Blahopřeji, že jste to skrz tuto náročnou kapitolu zvládli až na její konec.

4.3. Suspend a Resume

Vlákno může být explicitně „suspendováno“ a znovu uvedeno do chodu pomocí metod **Thread.Suspend** a **Thread.Resume**. Tento mechanismus je naprosto oddělený od blokování – oba systémy mohou pracovat paralelně a nezávisle na sobě.

Jedno vlákno může suspendovat samo sebe nebo jiné vlákno. Volání **Suspend** vyústí ve vstoupení vlákna do stavu *SuspendRequested* („požadována suspendace“) a jakmile to bude pro garbage collector vhodné, přenastaví se na *Suspended*. Z tohoto bodu může být vlákno opět uvedeno do provozu jen tím, že jiné vlákno zavolá metodu **Resume**. **Resume** funguje jen na suspendovaná vlákna, ne na zablokovaná.

Od dob .NET 2.0 jsou **Suspend** a **Resume** málo používané kvůli nebezpečí v některých situacích. Pokud budeme pracovat s nějakými veledůležitými (třeba systémovými) resources a jejich vlákno bude suspendováno, celá aplikace (nebo i počítač) se může dostat do deadlocku.

Ale bezpečné je zavolat **Suspend** na nějaké vlákno pomocí jednoduchého synchronizačního mechanismu – zavoláme **Suspend** na vlákno *A*, pak čekáme, než ho třeba hlavní vlákno probudí. Problém je ale v testování, jestli je vlákno *A* suspendováno, nebo ne:

```
static void Main()
{
    worker.NextTask = "MowTheLawn";
    if ((worker.ThreadState & ThreadState.Suspended) > 0)
        worker.Resume;
    else
        // Nemůžeme zavolat Resume, protože vlákno není
        // suspendováno
        // Místo toho přenastavíme proměnnou
        worker.AnotherTaskAwaits = true;
}
```

Tento postup je příšerně thread-unsafe. Může dojít k jeho narušení kdekoliv mezi pouhými pěti řádky, které jsme si napsali. Přestože by se to dalo různě ošetřit, bylo by to mnohem složitější než různé alternativy (jako třeba *AutoResetEvent* a **Monitor.Wait**). Tato skutečnost dělá **Suspend** a **Resume** naprosto neužitečnými na všech frontách, uvedli jsme si je tu jen pro úplnost.

4.4. Metoda Abort

Vlákna můžeme násilně ukončit právě pomocí metody **Abort**:

```
class Abort
{
    static void Main()
    {
        Thread t = new Thread(delegate() { while (true);});    // Zacyklení
        t.Start();
        Thread.Sleep(1000); // Necháme to vteřinku běžet...
        t.Abort();  // a pak ukončíme
    }
}
```

Vlákno, které se chystáme ukončit, přejde okamžitě po zavolání **Abort** do stavu *AbortRequested*. Pokud se pak bez problémů ukončí, přejde do stavu *Stopped*. Na tuto situaci můžeme počkat třeba pomocí **Join**:

```
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
    }
```

```
}
```

Po zavolání **Abort** se na cílovém vlákně vytvoří výjimka *ThreadAbortException*. Můžeme ji zachytit, ale pak se tato výjimka znovu vytvoří na konci *catch* bloku (aby se zajistilo, že vlákno skutečně skončí v pořádku). Znovuvyvolání můžeme zabránit zavoláním **Thread.ResetAbort** někde uvnitř *catch* bloku, vlákno pak znovu vstoupí do stavu *Running*, ze kterého ho znovu můžeme pomocí **Abort** dostat do stavu *AbortRequested*. V následujícím příkladu vyvoláme vlákno z mrtvých zpět mezi živé vždy, když se pokusíme zavolat **Abort**:

```
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("Já neumřu!");
        }
    }
}
```

Výjimka *ThreadAbortException* má jednu důležitou zvláštnost: pokud ji neošetříme, aplikace nespadne (na rozdíl od všech ostatních výjimek)!

Abort vám bude fungovat na vlákně v téměř jakémkoliv stavu – ať už normálně pracuje, je zablokované, suspendované nebo třeba zastavené. Pokud **Abort** zavoláme na suspendované vlákno, vznikne výjimka *ThreadStateException* (název na první pohled podobný *ThreadAbortException* probrané před chvílí, neplést!) na volajícím vlákně a proces přerušování vlákna pomocí **Abort** bude pokračovat teprve až nebude suspendováno. Takhle to funguje:

```
try { suspendedThread.Abort(); }
catch (ThreadStateException) { suspendedThread.Resume(); }
// Teď se suspendThread ukončí
```

4.4.1. Problémy s Thread.Abort

Možná si myslíte, že pokud vlákno nezavolá **ResetAbort**, můžeme očekávat, že se po zavolání **Abort** ukončí poměrně rychle. Ovšem existuje několik faktorů, které mohou držet vlákno ve stavu *AbortRequested* docela dlouhou dobu:

- Statické konstruktory nejsou nikdy ukončeny v polovině jejich kódu – vždy se musí dokončit celý kód konstruktoru.
- Stejně tak nikdy nedojde k utnutí kódu v *catch/finally* blocích.

- Pokud je zavoláno **Abort** a vlákno zrovna spouští unmanaged kód, dojde k ukončení teprve až se vlákno znovu dostane k managed kódu.

Poslední faktor nám může způsobit problémy, protože sám .NET Framework často využívá unmanaged kód a to po dlouhou dobu. Příkladem unmanaged tříd jsou různé třídy pro práci se sítí nebo s databázemi. Pokud máme server s pomalou odezvou, můžeme v oblasti unmanaged kódu zůstat klidně i několik minut (to samozřejmě závisí na samotném serveru, složitosti požadavku, …).

Používání **Abort** v kombinaci s čistým managed kódem je bez větších problémů, pokud používáme *using*, anebo ve *finally* bloku voláme **Dispose**. I tak jsme ale pořád ohroženi nějakými nepříjemnými překvapeními, podívejte se na tento kód:

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

Tento zápis pro vás asi není nic nového, kompilátor si ho převede do takovéto podoby:

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

A tady je ten problém. Může se totiž stát, že se **Abort** zavolá ve chvíli, kdy je sice instance *StreamWriteru* už vytvořená, ale před tím, než stihne aplikace spustit *try* blok. Pokud bychom se podívali na IL kód, zjistili bychom, že to samé se může stát i zatímco se instance *StreamWriteru* přiřazuje do proměnné *w*:

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

Ať už nastane jakýkoliv problém z předešlého odstavce, Dispose ve *finally* bloku se obejde a zůstane nám „prázdná“ instance *StreamWriteru*, která zabrání všem dalším pokusům o vytvoření souboru myfile.txt, dokud neskončí samotná aplikační doména.

Vyvstává tedy otázka, jak vlastně správně napsat „abort-friendly“ metodu. Nejběžnějším způsobem je ten, že vůbec **Abort** v metodě nezavoláme, místo toho jen nastavíme proměnnou, která indikuje, že by metoda **Abort** měla být zavolána. Pracovní vlákno pak periodicky kontroluje stav té proměnné a pokud je *true*, zavolá se **Abort**. Vše bude ještě lepší, pokud pracovní vlákno zavolá **Abort** samo na sebe – díky tomu je vlákno „abortnuto“ hned po skončení *try/finally* bloků:

```
class ProLife
{
    public static void Main()
    {
        Worker w = new Worker();
```

```
        Thread t = new Thread(w.Work);
        t.Start();
        Thread.Sleep(500);
        w.Abort();
    }

    public class Worker
    {
        // volatile zajistí, že proměnná abort nebude cachována
        volatile bool abort;

        public void Abort() { abort = true; }

        public void Work()
        {
            while (true)
            {
                CheckAbort();
                try { OtherMethod(); }
                finally { /* jakýkoliv potřebný úklid */ }
            }
        }

        void OtherMethod()
        {
            // Nějaká práce...
            CheckAbort();
        }

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

4.5. Ukončování aplikačních domén

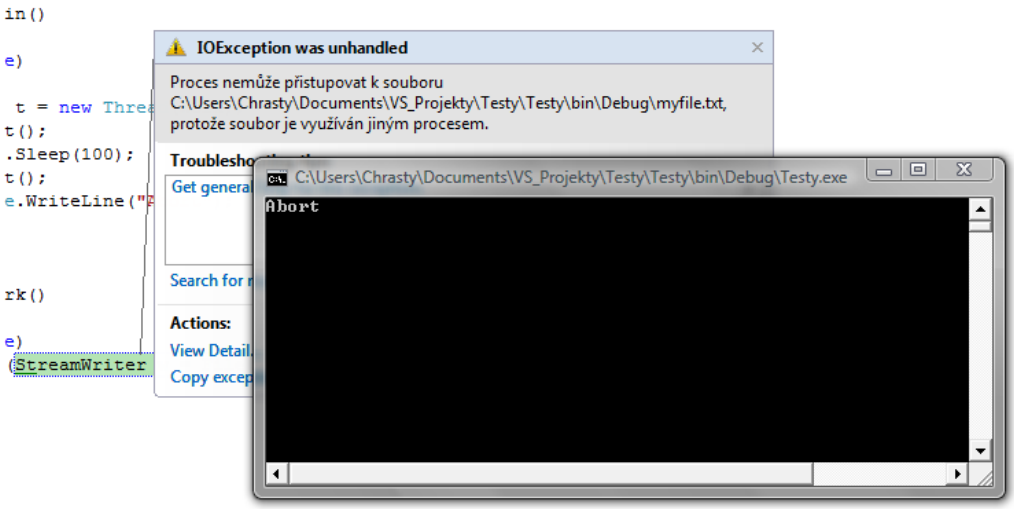
Dalším způsobem, jak zajistit bezpečnost volání **Abort**, je mít požadované vlákno v jeho vlastní aplikační doméně. Po zavolání **Abort** se totiž ukončí celá doména, společně se všemi instancemi, resources, … které nebyly správně ukončeny (případ *StreamWriteru* o několik řádků výše).

Abych pravdu řekl, je volání **Abort** zbytečné, protože když se ukončí aplikační doména, všechna vlákna se ukončí také (pomocí automatického zavolání **Abort**). Spoléhat se na to má ale jednu nevýhodu. Pokud bude volání **Abort** dlouho trvat (třeba kvůli dlouhému kódu v nějakém *finally* bloku), aplikační doména se neukončí a místo toho se vytvoří výjimka *CannotUnloadAppDomainException*. Z tohoto důvodu je lepší před ukončením aplikační domény zavolat explicitně **Abort** a **Join** s nějakým timeoutem, který si sami specifikujeme.

V následujícím příkladu vstupuje pracovní vlákno do nekonečného cyklu, kde pořád dokola vytváří a zavírá soubor pomocí abort-unsafe metody **File.CreateText**. Hlavní vlákno opakovaně vytváří a abortuje zmíněná pracovní vlákna. Aplikace většinou spadne po jedné nebo dvou iteracích, kvůli **CreateText** metodě, kterou přerušíme v půlce její práce:

```
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("Abort");
        }
    }

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



Teď si ukážeme stejný program, jen upravený, aby pracovní vlákno běželo ve své vlastní aplikační doméně, která je zrušena (metodou **Unload**) po ukončení vlákna. Aplikace poběží bez chyb, protože se „poškozená“ aplikační doména s neplatným odkazem na soubor vždy zruší.

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

class Program
{
    static void Main(string[] args)
    {
        while (true)
        {
            AppDomain ad = AppDomain.CreateDomain("worker");
            // Lambda výraz v kódu je ekvivalentní k:
            // Thread t = new Thread (delegate() { ad.DoCallBack (Work);
});
            Thread t = new Thread(() => ad.DoCallBack(Work));
```

```
            t.Start();
            Thread.Sleep(100);
            t.Abort();
            if (!t.Join(2000))
            {
                /* Vlákno ještě neskončí i přesto, že jsme zavolali Abort
                 * Zde můžete umístit další kód dle potřeby */
            }
            AppDomain.Unload(ad);   // Zničíme rozbitou doménu
            Console.WriteLine("Aborted");
        }
    }

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

Výstup bude podle všech očekávání: tisknutí textu „Aborted“ pořád dokola.

Vytváření a ničení aplikačních domén je ve světě počítačů považované za poměrně dlouho trvající operaci, trvá totiž i několik milisekund.

4.6. Ukončování procesů

Poslední situací, kdy může vlákno přestat existovat, je při ukončení rodičovského procesu. Taková situace nastane, pokud pracovnímu vláknu řekneme, že má běžet na pozadí (Vzpomínáte? Pomocí vlastnosti *IsBackground* nastavené na *true*.) a hlavní vlákno dokončí svoji práci. Pracovní vlákno pak není schopné udržet aplikace při životě (je ignorováno, běží přece někde na pozadí), ukončí se proces aplikace a pracovní vlákno se díky tomu ukončí také.

Když je vlákno ukončeno kvůli konci rodičovského procesu, je v tu ránu mrtvé, ani žádné *finally* bloky se neprovedou. Ke stejné situaci dojde, pokud aplikaci ukončíme přes Správce úloh (Ctrl + Shift + Esc) nebo přes metodu **Process.Kill**.

Vypadá to, že jsme na konci. Tím myslím úplně. Na několika desítkách stran jsme se podívali na základy i na pokročilá témata ohledně vláken a řekli jsme si o spoustě tříd. Některé z nich sice dnes využití nenajdou, ale je dobré o nich vědět, nemyslíte?