KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

SYSTÉMOVÉ PROGRAMOVÁNÍ V JAZYCE C#

ALEŠ KEPRT

VÝVOJ TOHOTO UČEBNÍHO TEXTU JE SPOLUFINANCOVÁN
EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY

Olomouc, 30.4.2008

Abstrakt

Tento studijní text se věnuje základům systémového programování na platformě .NET a především má za cíl posloužit jako studijní text k praktickým cvičením předmětů Operační systémy, Systémové programování a Programové vybavení počítačů ve všech jejich semestrech a formách. Látka přímo navazuje na kapitoly přednášené v Operačních systémech a jako programovací nástroj používá jazyk C# na platformě .NET. Text je zároveň použitelný i pro cvičení v jazyce Visual Basic, kde řada témat je identických a u vybraných pasáží, kde by studenti mohli mít problémy, je Visual Basicu věnována zvláštní pozornost přímo v textu.

Cílová skupina

Tento studijní materiál je primárně určen pro studenty oboru Aplikovaná informatika uskutečňovaného v kombinované formě na Přírodovědecké fakultě Univerzity Palackého v Olomouci a pro studenty oborů učitelství pro střední školy v kombinacích s výpočetní technikou uskutečňovaných v prezenční formě tamtéž. Studenti těchto oborů mají programování v jazycích C# a Visual Basic jako povinné součásti prvních ročníků studia, takže práce s tímto textem pro ně bude snadná a srozumitelná. Text je vhodný také pro studenty oborů Informatika a Aplikovaná informatika uskutečňovaného v prezenční formě tamtéž, kteří jako primární jazyk používají C++ a jazyk C# mají jako volitelný předmět. Kapitoly obsažené v textu pokrývají zejména látku předmětů XOSY/Operační systémy, XOS/Operační systémy, XSYS/Systémový software, OS2AI/Operační systémy 2 a VP1AW/Programové vybavení počítačů 1, částečně pak také látku předmětů JCSBI/Jazyk C# a PNBI/Platforma .NET.

Další informace k teorii týkající se zde probírané látky je možno nalézt v učebním textu Operační systémy [Kep07].

Obsah

Studijní plán 6
1 Procesy a vlákna 7
1.1 Procesy 7
1.1.1 Parametry spouštěného procesu 7
1.1.2 Shell Execute 8
1.1.3 Metody třídy Process 8
1.1.4 Přesměrování standardního vstupu a výstupu 9
1.1.5 Ukončení procesu 10
1.2 Vlákna 11
1.2.1 Základní prvky třídy Thread 11
1.2.2 Zásobník 12
1.2.3 Běh a stavy vlákna 12
1.2.4 Thread Local Storage (TLS) 13
1.2.5 Ukončení vlákna 14
1.2.6 Další studium 15
1.3 Priority 15
2 Aplikační domény 17
2.1 Třída AppDomain 17
2.2 Vytvoření a zrušení domény 17
2.3 Načtení kódu do domény 18
2.4 Volání kódu mezi doménami 19
2.5 Systémové události v doménách 19
2.6 Izolace mezi doménami 21
3 Synchronizace vláken a procesů 22
3.1 Přehled 22
3.2 Monitor a zámek 22
3.2.1 Zámek pro vzájemné vyloučení 23
3.3 Mutex 25
3.4 Princip čekatelných objektů 25
3.5 Semafor 26
3.6 Vláknová afinita (thread affinity) 26
3.7 Signál 27
3.7.1 Třída EventWaitHandle 27
3.7.2 Signalizace pomocí monitorů 27
3.8 Čtenáři a písaři 28
3.8.1 Třída ReaderWriterLockSlim (.NET 3.5) 28
3.8.2 Třída ReaderWriterLock 29

3.9 Blokované (interlocked) operace 32
3.9.1 Základní aritmetika 33
3.9.2 Složitější aritmetika 33
4 Asynchronní výpočetní techniky 35
4.1 Přehled 35
4.2 Fond vláken (ThreadPool) 35
4.2.1 Úlohy na pozadí 35
4.2.2 Asynchronní čekání 36
4.2.3 Konfigurace fondu 36
4.3 Časovače (třikrát jinak) 37
4.3.1 Třída System.Windows.Forms.Timer 38
4.3.2 Třída System.Threading.Timer 39
4.4 Vzor asynchronní aktualizace GUI 40
4.5 Asynchronní programový model (APM) 41
4.5.1 Co je to APM 41
4.5.2 Použití APM 42
4.5.3 Více o asynchronní práci se soubory 44
4.5.4 Asynchronní delegáty 44
4.6 BackgroundWorker (.NET 2.0) 44
4.6.1 Popis 44
4.6.2 Použití 45
4.7 Roury 45
5 Paměť a zdroje 48
5.1 Pojem správy paměti 48
5.2 Automatická správa paměti v .NETu 48
5.2.1 Organizace malé řízené haldy 49
5.2.2 Pinning 50
5.2.3 Hledání dožitých objektů 50
5.2.4 Správa velké haldy 50
5.2.5 Explicitní spuštění kolektoru 50
5.2.6 Tři verze kolektoru 51
5.3 Životní cyklus objektu 51
5.3.1 Obyčejné referenční třídy 51
5.3.2 Třídy používající neřízené zdroje 52
5.3.3 Vliv finalizeru na život a resurekce (oživování mrtvých) objektů 53
5.3.4 Třída agregující objekty používající neřízené zdroje 53
5.3.5 Specifika C++/CLI 53
5.4 Další témata 55

5.4.1 Chování systému při nedostatku paměti 55
5.4.2 Paměťová brána 55
A Windows – kapitola nula 58
A.1 Windows API 58
A.2 Výsledkové kódy 59
A.3 Identifikátory objektů (hendly) 59
A.4 Znaky a texty 60
B Thread Local Storage 63
B.1 Úvod 63
B.2 Kontext – lokální paměť vlákna bez podpory operačního systému 63
B.3 TLS s podporou operačního systému 64
B.3.1 Windows API (Windows 95/98/NT/2000/XP/Vista) 64
B.3.2 POSIX a Linux 65
B.3.3 Platforma .NET 66
B.4 Podpora TLS v překladačích jazyků 66
B.4.1 Visual C/C++ 66
B.4.2 GNU C/C++ (GCC) a Sun Studio C/C++ 67
B.4.3 Jazyky v prostředí .NET Framework 67
B.5 Případová studie: BiF 67
B.5.1 Co je to BiF 67
B.5.2 Paralelizace 67
Reference 69

Studijní plán

Tento studijní text je zaměřen velmi prakticky a od studentů se předpokládá, že jednotlivá probíraná témata si budou průběžně zkoušet na programovacích úlohách. Ty je možno realizovat buď v jazyce C#, Visual Basic nebo i v jiném jazyce pracujícím v prostředí .NET. Text je zaměřen primárně na jazyk C# a v několika místech, která by mohla uživatelům Basicu dělat problémy je uveden i příslušný kód v tomto jazyce. Použití jiných jazyků je pak čistě na schopnostech studentů, ale mělo by to být možné, neboť se budeme zabývat většinou obecnými koncepty a voláním knihovny BCL (Base Class Library) knihovny .NET.

Odborná úroveň textu a složitost je střední až vysoká, text je určen pokročilejším programátorům a studentům ve vyšších ročnících studia informatiky a aplikované informatiky. Tento studijní text je také koncipován jako navazující či doplňující materiál k publikaci [Kep07], která se teoretickými otázkami operačních systémů a jejich programování podrobně zabývá. Od studentů tohoto textu se tedy předpokládá znalost příslušných kapitol textu [Kep07] a také schopnost pracovat ve vývojovém prostředí Visual Studio 2005 nebo 2008. (Některá probíraná témata se týkají .NETu 3.5 a je možno je implementovat jen ve verzi VS 2008.)

Studijní plán pro jednotlivé obory a předměty studia je dán vyučujícím, speciálně pak u prezenční formy studia. Pokročilejší (či odvážnější) studenti mohou i postupovat lineárně a projít všechny kapitoly a vyřešit všechna cvičení v textu. Pro školní potřeby však stačí projít ty kapitoly, které jsou ve studijním plánu studovaného předmětu.

Všichni studenti by se měli nejprve seznámit s přílohou A vysvětlující základy programování ve Windows, které jsou i pro programátory v .NETu důležité. Dále pak můžete postupovat podle některého z těchto doporučených plánu:

Doporučený plán pro XOSY, XOS a XSYS: Kapitoly 1, 3, 4.1–4.3 a 4.5.
Studium je zaměřeno na probrání práce s vlákny a synchronizace. Předpokládá se, že studenti již znají C#, práci s textem a soubory. Vynechávají se nejsložitější témata.

Doporučený plán pro VP1AW: Kapitoly 1 a 3.
Studium se omezuje na několik základních témat kolem vláken a synchronizace, která se dají stihnout ve velmi limitovaném čase. Předpokládá se, že studenti již znají C#, práci s textem a soubory.

Doporučený plán pro JCSBI: Kapitoly 1 a 2.
Pro tento předmět se použijí jen úvodní kapitoly, které se jako jediné kryjí s obsahem předmětu.

Doporučený plán pro PNBI: Kapitoly 1–5.
Tento předmět se zabývá pokročilejšími tématy a použijí se všechny kapitoly kromě látky již probrané v JCSBI.

Doporučený plán pro OS2AI: Podle potřeby.
Text se používá ve škole při hodinách cvičení, dle instrukcí vyučujícího.

Pro podrobnější studium zde probírané látky je možno doporučit [Duf06] či [Tro07].

1 Procesy a vlákna

Studijní cíle: Úvodní kapitola bude patřit procesům a vláknům. Studenti se naučí především založit objekty procesu a vlákna, což nemusí být pro začátečníky zrovna triviální úkol. V ukázkových příkladech si dále vyzkouší některé základní operace s nimi a v závěru kapitoly se podíváme ještě na oddělení globálních proměnných pomocí objektů a TLS.

Klíčová slova: proces, vlákno, TLS (Thread Local Storage)

Potřebný čas: 110 minut (plus čas k vypracování úloh)

1.1 Procesy

Jak víme, proces ve Windows vzniká tak, že spustíme nový EXE soubor. V .NETu to funguje úplně stejně. Základní třída reprezentující proces je `System.Diagnostics.Process`. (Do projektu si tedy přidáme `using System.Diagnostics;`.) Když chcete spustit nový proces, vytvoříte nejprve instanci této třídy, v ní nastavíte vše potřebné a pak proces spustíte. K právě běžícímu (tj. „svému") procesu se dostanete pomocí `Process.GetCurrentProcess()`. Jméno spouštěcího EXE souboru svého procesu získáte např. takto: `Process.GetCurrentProcess().MainModule.FileName`.

Průvodce studiem

Objekt procesu v .NETu není přímo objektem procesu operačního systému. Skutečné systémové procesy totiž obsahují mnoho hodno, které se často mění a bylo by neefektivní udržovat stále platnou kopii v .NETu. Proto je objekt procesu v .NETu pouze kopií informací o skutečném procesu v určitém čase. Při vytváření nového procesu proto stačí vytvořit prázdný objekt, pak u něj nastavit všechny požadované parametry a na závěr teprve spustit skutečný proces podle objektu .NETu.

1.1.1 Parametry spouštěného procesu

Objekt nového procesu vytvoříte konstruktorem bez parametru. Veškeré nastavení parametrů spouštěného procesu nastavíte až potom, pomocí property `Process.StartInfo` typu `ProcessStartInfo`. Tato struktura má mnoho prvků, podívejme se na ty nejdůležitější:

- `Arguments` – parametry (argumenty) na příkazové řádce (délka max. 2003 znaků)
- `CreateNoWindow` – true=neotevře se nové konzolové okno (výchozí je false). Toto je dobré, když přesměrováváte konzolový vstup a výstup.
- `FileName` – jméno spouštěcího souboru. (Při použití shell execute může být soubor libovolného asociovaného typu.)
- `UseShellExecute` – povoluje používání shell execute (výchozí je true)
- `Verb` – nastaví příkaz pro shell execute (výchozí je prázdný string "")
- `Verbs` – pole obsahující všechny zaregistrované verby (podle přípony souboru)

- `ProcessWindowStyle WindowStyle` – styl okna při spouštění. Může být `Normal`, `Hidden`, `Maximized` či `Minimized`.
- `WorkingDirectory` – pracovní adresář procesu

Průvodce studiem

Struktura `ProcessStartInfo` má ještě řadu dalších součástí. Podrobnosti najdete v [MSDN].

Proces lze spouštět i pod jiným uživatelem. Potřebujete k tomu pochopitelně znát uživatelské jméno a uživatelovo heslo. Takto spuštěný proces má především všechna práva a omezení, jako daný uživatel, takže jde o techniku vhodnou jen pro speciální situace. K nastavení spuštění procesu pod jiným uživatelem slouží tyto další položky ve `StartInfo`: `UserName`, `Password`, `LoadUserProfile`.

Průvodce studiem

*Nevíte-li, o co jde, mrkněte na nápovědu systémového programu **runas**.*

1.1.2 Shell Execute

ShellExecute umí spouštět i jiné soubory, než EXE.

Zastavme se nyní krátce u pojmu Shell Execute. Jak víme, ve Windows lze dvojklikem otevřít soubor i jiného typu než EXE. Spouštění takových „dokumentových" souborů funguje právě pomocí funkcionality zvané shell execute – v systémovém registru jsou zapsány přípony souborů známých typů a akce, která má být vykonána, když na soubor dvojkliknete (v Microsoftím slangu: „poklepáte" jej"). Dvojklik spustí akci „open" (viz položka `Verb`). Další obvyklá akce je třeba „print" = tisk souboru. Verb (anglicky: sloveso) je tedy jednoduše příkaz udávající, co se má se souborem stát.

Pokud spouštíte jen EXE soubory (a jste si jistí, že opravdu jde o EXE soubory), je lepší používání shell execute zakázat (nastavíte `UseShellExecute = false`), protože necháte-li jej zapnuté, nebudou fungovat některé speciální funkce procesů, se kterými chceme dále pracovat.

Průvodce studiem

Další podrobnosti k systému shell execute najdete v [MSDN]. Přečtěte si nápovědu k systémové funkci `ShellExecuteEx`, kde je programové chování tohoto systému popsáno.

1.1.3 Metody třídy Process

Zde je popis nejdůležitějších metod třídy `Process`:

- `CloseMainWindow()` – ukončí proces posláním zprávy Close do hlavního okna

- `Close()` a `Dispose()` – uvolní neřízené prostředky → pozor, `Process` je IDisposable
- `GetCurrentProcess()` – vrací objekt aktuálního procesu (statická metoda)
- `Kill()` – násilně ukončí proces
- `Start()` – spustí proces
- `WaitForExit()` – čeká na ukončení procesu
- `WaitForInputIdle()` – čeká, až proces zpracuje všechny zprávy (ve smyčce zpráv)

Metoda `Start()` spouštějící proces existuje v několika variantách. Kromě základní varianty bez parametrů máme k dispozici ještě několik variant statických s různými parametry. Tyto statické metody samy vytvoří objekt procesu, spustí jej a vrací objekt typu Process. Možné parametry metody jsou

- Struktura ProcessStartInfo
- Jméno souboru, nepovinně je možno přidat parametry
- Totéž plus jméno uživatele, jeho doména a heslo (v kódované formě)

S metodami `CloseMainWindow()` a `Kill()` zacházejte opatrně. Prvně jmenovaná totiž funguje jen v ideálních podmínkách, neboť posílá zprávu Close oknu a chování při této zprávě je závislé čistě na každém programu a k zavření okna tedy dojít nemusí. `Kill()` pak ukončuje proces násilně, lidově řečeno „odstřelí jej“, čímž přeskočí veškerý kód, který by se při běžném ukončení vykonal. Funguje u všech procesů, včetně konzolových, ale výsledkem může být mnoho problémů (nekonzistentní soubory na disku, pokud je ukončený proces právě měnil či vytvářel atp.). Další informace o těchto dvou metodách najdete v [MSDN].

Metoda `WaitForInputIdle()` se týká jen okenních aplikací. Její význam je při startu procesu. Je vhodné ji zavolat ihned po spuštění procesu, čímž de facto počkáme na dokončení inicializace toho procesu, neboť při spuštění se nejprve provádí nějaký (nám neznámý) startovací kód a potom se teprve začnou zpracovávat zprávy. Jakmile je fronta zpráv prázdná, máme jistotu, že proces již je plně inicializován.

1.1.4 Přesměrování standardního vstupu a výstupu

Vstup i výstup lze přesměrovat.

Dalším tématem, které nás zajímá, je směrování standardního vstupu a výstupu, které v .NETu funguje stejně jako v operačním systému. Používá se tedy především u konzolových aplikací k přesměrování výstupu z konzoly do souboru či na vstup jiného procesu, k přesměrování vstupu z klávesnice na soubor či výstup jiného procesu či k přesměrování chybového výstupu do souboru či na vstup jiného procesu. Můžete tedy přesměrovat všechny tři standardní streamy (proudy). Jsou k tomu tři položky ve `StartInfo`:

`RedirectStandardOutput`, `RedirectStandardInput`, `RedirectStandardError`

Jsou to přepínače (true/false) a výchozí nastavení je u všech tří false. Nastavíte-li některý z nich na true, potom máte k dispozici tři streamy ve třídě `Process`:

`StandardOutput`, `StandardInput`, `StandardError`

Při přesměrování je vhodné dodržet tento postup:

1. Nastavíme přepínač(e) přesměrování na true.
   `StartInfo.RedirectStandardOutput = true;`
2. Spustíme proces.
   `Process.Start();`

3. Zapisujeme do vstupního proudu.
   `Process.StandardInput.WriteLine("text");`
4. Čekáme na ukončení procesu.
   `Process.WaitForExit();`
5. Čteme z výstupního a/nebo chybového proudu.
   `Process.StandardOutput.ReadToEnd();`

Zde popsaný postup v pěti krocích si procvičíte v následující úloze.

Úloha 1. Procesy

Vyzkoušíte si spouštění procesu a přesměrování vstupu a výstupu. Pro toto testování vytvoříme jen jeden program, který se ale podle typu spuštění bude chovat jedním z dvou různých způsobů:

1. Při spuštění programu bez parametru přejde na funkci SpuštěníBezParametru(). Tato funkce spustí nový proces – opět tentýž EXE soubor, ale s parametrem "3". Navíc přesměruje novému procesu vstup i výstup. Na vstup mu dá string "Ahoj" (pomocí funkce WriteLine). Výstup převede na velká písmena a vypíše na konzolu.
2. Při spuštění programu s parametrem přejde na funkci SpuštěníSParametrem(args[0]). Tato funkce přečte text ze standardního vstupu (`Console.ReadLine`) a vypíše jej na obrazovku tolikrát, kolik je uvedeno v parametru příkazové řádky.

Tento program tedy při spuštění spustí ještě druhou svou kopii. Tato pak vypíše 3x AHOJ, ale toto se neobjeví na obrazovce. První kopie programu totiž zachytí tento text, převede jej na velká písmena a pak teprve vypíše. Tím bude ověřeno, že spuštění procesu a přesměrování funguje.

1.1.5 Ukončení procesu

Třída `Process` obsahuje opravdu velké množství dalších prvků, my se však podrobněji podíváme pouze na druhou důležitou věc, jíž je problematika ukončení procesu. Řádné ukončení procesu je pouze tehdy, když proces vykoná všechen kód a sám skončí (obvykle tedy proběhne a skončí startovací metoda Main()). Součásti související s ukončením procesu jsou:

- `HasExited` – vrací příznak, že/zda proces již skončil.
- `ExitCode` – vrací kódu ukončení (hodnota main() v C++, v .NETu je to vždy 0).
- `StartTime` – vrací čas spuštění procesu.
- `ExitTime` – vrací čas ukončení procesu.
- `WaitForExit()` – čeká, až proces skončí. Lze zadat časový limit doby čekání.
- `Exited` – tato událost je vyvolána v okamžiku skončení procesu.
- `Kill()` – násilně proces odstřelí, vynechá při tom ukončovací kód procesu. Odpovídá funkci `TerminateProcess()` z Windows API.

Stav procesu můžeme také sledovat (v jiném procesu), potřebujeme k tomu jen mít objekt procesu. Ten pochopitelně máme především v tom procesu, který sledovaný proces vytvořil. (Čili v obvyklém případě rodičovský proces sleduje svého potomka, ale obecně tomu může být i jinak.)

Pro asynchronní sledování skončení procesu pomocí události `Exited` musíme nejprve nastavit property `EnableRaisingEvents` na true. (Ve výchozím stavu je toto vypnuto, událost `Exited` tedy není při skončení procesu vyvolána.) Tato drobná práce navíc je vyžadována z důvodu úspory systémových zdrojů: Pokud se `Exited` nepoužívá, šetří se systémové zdroje (čekatelné objekty ve Windows).

1.2 Vlákna

Všechny věci kolem vláken jsou v jmenném prostoru `System.Threading`. Základní třída je `System.Threading.Thread`. Jak víme, každý proces alespoň jedno vlákno vždy má – to je to, kterým se program startuje. Objekt tohoto vlákna můžete jednoduše získat pomocí public property `Thread.CurrentThread` (je to statická položka třídy `Thread` a vrací objekt vlákna).

Další vlákna pak můžeme vytvářet jako nové objekty typu `Thread`:
`Thread další_vlákno = new Thread(ThreadProc);`

Vytváření vláken funguje podobně jako u procesů tak, že nejprve vytvoříme objekt vlákna v .NETu a potom zavoláním jeho metody `Start()` necháme vytvořit a spustit skutečné vlákno v operačním systému.

Zde `ThreadProc` je nějaká vaše metoda, která bude použita jako startovací metoda vlákna. Tato metoda může mít nepovinně jeden libovolný parametr a nic nevrací. (Volitelným parametr slouží k tomu, když potřebujeme spouštěnému vláknu předat nějaké parametry, obvykle pro upřesnění jeho úkolu či identity.) Podobně jako u procesů, objekt typu `Thread` je jen obrazem skutečného vlákna v operačním systému. Skutečné vlákno v systému vznikne až při volání `Start()` a po skončení této metody vlákno v operačním systému zanikne, zatímco objekt vlákna v .NETu si můžeme ponechat i déle.

1.2.1 Základní prvky třídy Thread

- `Start()` – Spustí vlákno.
- `Sleep()` – Uspí vlákno na daný čas. Je to statická metoda, takže vlákno může takto uspat jen samo sebe. Čas lze zadat v milisekundách nebo jako hodnotu typu `TimeSpan` – ta se však také zaokrouhlí na celé milisekundy. A skutečná doba spaní závisí na konkrétní verzi Windows, jak přesně dokáže čas měřit.
- `Priority` – Zjistí či změní prioritu vlákna. Povolené hodnoty jsou výčtového typu `ThreadPriority` a je zde jen 5 možností: `Lowest`, `BelowNormal`, `Normal`, `AboveNormal`, `Highest`. Priorita vlákna je relativně vztažená k prioritě procesu.

Úloha 2. Vlákno

Vyzkoušíte si nejjednodušší práci s vlákny. Vytvořte třídu `Písař`, která bude zabalovat vlákno (tj. každý objekt této třídy bude novým vláknem).

Vlákno bude dělat jednoduchou věc: Ve smyčce bude vypisovat postupně 100x nějaký znak na obrazovku a vždy po napsání znaku čekat 1ms. Znak, který má být vypisován, bude nastavitelný v konstruktoru.

V mainu si pak vytvořte tři instance této třídy, pokaždé s jiným znakem. A všechny spusťte. Může to vypadat třeba takto:

```
Písař p1 = new Písař('#');
Písař p2 = new Písař('@');
Písař p3 = new Písař('$');
p1.Start();
p2.Start();
p3.Start();
```

Program pak při spuštění na obrazovku vypisuje znaky z jednotlivých vláken a ty se nepravidelně střídají. Parametr předávaný do konstruktoru třídy Písař je možno předat jako volitelný parametr do spouštěcí metody vlákna, nebo si jej uložíme do proměnné v objektu typu Písař.

1.2.2 Zásobník

Jak víme, každé vlákno v operačním systému má svůj programový zásobník. Vlákna v .NETu se chovají jako běžná vlákna ve Windows, proto nepřekvapí, že mají také standardní zásobník o velikosti 1MB. Tuto velikost však lze změnit v nastavení překladače, to je metoda vhodná hlavně pro nastavení prvního vlákna aplikace, nebo uvedením požadované velikosti zásobníku v konstruktoru třídy Thread. Minimální velikost je 128KB a doporučuje se standardní velikost 1MB zachovat, pokud nepotřebujeme vytvářet velké množství vláken. (Např. pro 1000 vláken již zásobníky zaberou 1GB paměti, tak velký počet vláken v jediném programu je však raritou.)

Úloha 3. Zjištění velikosti rámce volání funkce

Jak víme z přednášek, při volání funkcí se na zásobníku vytváří tzv. rámec volání. Je to abstraktní pojem vyjadřující, že s jedním konkrétním voláním funkce je spojena určitá část programového zásobníku. Pro jednoduchost zde budeme rámcem nazývat skutečně celou oblast zásobníku použitou při jednom volání, vaším úkolem v této úloze je zjistit jeho velikost.

Kdybychom pracovali v jazyce C, pak můžeme odpověď najít velmi snadno, stačí ve dvou sousedních voláních zjistit hodnotu registru `esp` a odečíst je od sebe. Kód by vypadal přibližně takto (hodnotu `esp` ve skutečnosti zjistíme pomocí assembleru, zde zjednodušeno):

```
int první() { return esp - druhé(); }

int druhé() { return esp; }
```

V .NETu toto udělat nemůžeme, protože .NET nepracuje přímo s pamětí a neexistuje v něm pojem adresa. I přesto však je úloha řešitelná, použijte následující pomůcku: Při vytvoření nekonečné rekurze program po zaplnění zásobníku spadne s výjimkou `StackOverflowException`. To je pochopitelné, během vnořování však budete inkrementovat hodnotu nějaké statické proměnné a v okamžiku zaplnění zásobníku budete díky tomu vědět, do jaké úrovně vnoření se vlákno dostalo. Potom přidáte do rekurzivní metody nějakou proměnnou známé velikosti (například klasický 4bajtový int) a běh zopakujete. Tentokrát každé volání zabere o 4 bajty více, takže se zásobník zaplní již při nižší úrovni vnoření.

Pomocí dvou takto změřených hodnot pak dokážete říci, kolik bajtů přesně zabírá jedno volání funkce. Tato hodnota se může lišit podle nastavení překladače a typu rekurzivní funkce, vyzkoušejte pro různé situace! Všimněte si, že výsledná hodnota bude vždy násobkem čtyř, protože jak víme data jsou v programovém zásobníku ukládána vždy ve 4bajtových buňkách.

Průvodce studiem

Tato úloha byla poněkud obtížnější. Pokud se vám nepodařilo ji vyřešit, nezoufejte, správný postup bude předveden na hodině ve škole.

1.2.3 Běh a stavy vlákna

Property `IsAlive` můžeme použít ke zjištění, zda je běžící vlákno ještě „živé“, tj. zda již začalo a ještě neskončilo. Přesnější informaci o stavu vlákna získáme z property `ThreadState`.

Průvodce studiem

Property `ThreadState` *je jen pro čtení a týká se jen objektu vlákna. V některých situacích se může tato hodnota lišit od stavu vlákna v operačním systému, je to z toho důvodu, že systém obecně nepodává informace o tom, co přesně s vlákny dělá.*

Vlákno může být v jednom čí více z následujících stavů (kombinace stavů jsou povoleny, ne však všechny, protože některé kombinace nedávají smysl):

- `Unstarted` – před voláním Start(), čili k objektu vlákna není přiřazeno vlákno v systému
- `Running` – vlákno je zařazeno v běhovém režimu operačního systému
- `Aborted` – vlákno bylo přerušeno pomocí `Abort()`, stejný stav jako `Stopped`
- `AbortRequested` – z jiného vlákna je požadavek na `Abort()`
- `Stopped` – vlákno skončilo
- `Suspended` – vlákno bylo pozastaveno voláním `Suspend()` (pro ladicí účely)
- `SuspendRequest` – je požadavek na `Suspend`
- `WaitSleepJoin` – vlákno čeká v řízeném kódu po volání `Wait()`, `Sleep()` nebo `Join()`
- `Background` – vlákno je zařazeno k běhu na pozadí

Pokud bychom chtěli testovat, zda je vlákno zařazeno k běhu, pomocí dotazu na stav `Running` (příkazem `if(thread.ThreadState == ThreadState.Running)` …) nebude to fungovat, neboť hodnoty `ThreadState` jsou bitové kombinace a kromě zde uvedených hodnot může být vlákno i v některém z dalších interních stavů zde neuvedených, stav `Running` má přitom číslo nula. Test, zda vlákno běží, tedy musíme provádět pomocí výše zmíněné property `IsAlive`, nebo pomocí dotazu, zda vlákno není ve stavu `Unstarted` či `Stopped`.

Další informace o stavech vlákna najdete v [MSDN] na stránce `ThreadState`.

1.2.4 Thread Local Storage (TLS)

Vlákna v jednom procesu sdílejí paměťový prostor, paměť je tedy společná všem vláknům v procesu. Někdy se však hodí, aby vlákno mělo nějakou vlastní paměť. Díky objektově orientovanému přístupu mohou vlákna mít de facto vlastní paměť jednoduše tak, že si vytvoří své objekty. Je-li na začátku kódu vlákna vytvoření objektu (operátorem `new`), pak každé vlákno bude mít tento objekt jiný.

Další alternativou je použít TLS, což je prostředek umožňující nastavit platnost statických proměnných na rozsah vlákna. TLS se tedy týká jen proměnných na globální úrovni (v jazyku C# označených jako `static`, v Basicu jako `shared`). TLS je paměť omezené velikosti, která patří jen jednomu vláknu. Proměnné umístěné v TLS jsou přístupné odkudkoliv, stejně jako běžné statické proměnné, ale jsou v každém vlákně jiné. Proměnnou do TLS lze umístit voláním `Thread.SetNamedDataSlot` / `Thread.GetNamedDataSlot` nebo použitím atributu `ThreadStatic` (atribut se uvede v hranatých závorkách před definicí proměnné).

Podrobně je toto rozepsáno v samostatném článku [Kep05], jehož přepis najdete v příloze B.

Úloha 4. TLS

Otestování TLS na nějaké jednoduché úloze je dosti problematické – vznikne totiž docela krkolomný kód. Nicméně nějak to vyzkoušet potřebujete, tak se neděste tomuto zadání…

Vyjdeme z předchozí úlohy a upravíme ji takto:

- Původní třídu `Písař` přejmenujte na název `Vlákno`.
- Vytvoříte novou třídu `Písař` s jedinou statickou metodou jménem `Piš`. Metoda bude bez parametru a pouze vypíše jeden znak, který si najde v TLS pomocí volání `GetNamedDataSlot("znak")`.
- Třída vlákno pak místo vypisování znaku (jak to bylo v úloze 1) bude volat onu statickou metodu `Písař.Piš`.
- Dopište zbytek kódu, aby aplikace fungovala stejně jako v úloze 1.

Vyzkoušejte i druhou alternativu téhož programu s atributem `ThreadStatic`.

Zkuste použít ještě jednu další novou věc: Ve třídě `Thread` lze použít i spouštěcí metodu s parametrem. Použijte ji na předání znaku k vypisování. Znamená to, že znak nebudete již předávat v konstruktoru, ale až v metodě Start. Ta totiž spustí vlákno a předá mu onen parametr. Použijete k tomu metodu `Thread.Start(object)` a konstruktor `Thread(ParametrizedThreadStart)` – popis najdete v [MSDN].

1.2.5 Ukončení vlákna

Vlákno může skončit třemi způsoby:

a) Skončením spouštěcí metody – toto je obvyklý způsob ukončení vlákna.
b) Vyhozením výjimky `ThreadAbortException`. Tuto výjimku vyhodíte voláním `Thread.CurrentThread.Abort()` – voláte to takto na svém vláknu.
c) Voláním metody `Abort()` na jiném vlákně – tímto vynutíte ukončení onoho vlákna. Ukončování jiných vláken je velmi choulostivá věc a může to zle dopadnout – důrazně doporučuji přečíst poznámky (remarks) v MSDN u metody `Thread.Abort()`.

Bez ohledu na důvod ukončení nějakého vlákna, jiné vlákno může čekat na „dokončení toho ukončení“. ☺ Metoda `Thread.Join(vlákno)` pozastaví běh aktuálního vlákna, dokud vlákno dané jako parametr neskončí. Používá se to obvykle tam, kde víme, že druhé vlákno dělá nějaký job, který má brzo skončit, a chceme či potřebujeme na něj počkat. Tato metoda je tedy obdobou metody `Process.WaitForExit()`, kterou již známe.

Poznámka: Volání `Thread.Abort()` nemusí úplně vždycky způsobit ukončení vlákna. Před vlastním ukončením vlákna jmenovanou výjimkou se totiž vykoná kód všech nadřazených `finally` bloků. Pokud by tento kód měl třeba nekonečnou smyčku, vlákno nemůže nikdy skončit.

Poznámka: Metodu `Thread.Abort()` považujte za násilné ukončení vlákna (tj. zabití – kill). Správný postup ukončení vlákna je bez použití této metody, tj. například pomocí nějaké proměnné typu **bool**, jejímž nastavením na true bude hlavní vlákno signalizovat žádost o ukončení všech pracovních vláken. Každé vlákno pak musí testovat tuto proměnnou a při zjištění hodnoty true se samo ukončit. Žádný jiný způsob ukončování vláken, než když vlákna sama skončí, není doporučen.

Po skončení všech vláken v procesu je proces ukončen. Tedy přesně v okamžiku, kdy poslední běžící vlákno skončí, tak skončí i celý proces. Zvláštní postavení však mají vlákna běžící na pozadí, která sama o sobě nevynucují pokračování procesu. Tato vlastnost se nastavuje (nebo zjišťuje) pomocí property `IsBackground` a jakmile v procesu zbývají již jen vlákna na pozadí, je také ukončen. Vlákno na pozadí má také nastaven příznak stavu `Background` v `ThreadState` (viz výše).

Další podrobnosti

Ukončení vlákna abortem je poměrně choulostivá operace, mohlo by při tom totiž dojít k poškození něčeho důležitého. Například ve `finally` blocích bývá kód, který má za úkol uklidit nepořádek po předchozím kódu. Přitom kód ve `finally` blocích obvykle nemůže sám způsobit žádnou chybu. Co by se však stalo, kdyby běžící `finally` blok byl přerušen abortem? Nedokončil by se a potřebný úklid by tedy nebyl proveden. Aby se zabránilo podobným problémům, .NET definuje tzv. abort delay regiony, což jsou úseky kódu, kde abort nemůže nastat. Jsou to jmenovitě všechny bloky `catch` a `finally`, veškerý neřízený kód a také constrained execution regions (CER, jen pro speciální systémové účely).

Výjimka `ThreadAbortException` při ukončování vlákna postupně probublá až do startovací metody, cestou vlákno vykonává všechny související `catch` a `finally` bloky. Dostane-li se vlákno cestou do jiné aplikační domény, výjimka je změněna na `AppDomainUnloadedException`. Více o aplikačních doménách se dozvíte v následující kapitole.

1.2.6 Další studium

Procesy i vlákna má ještě celou řadu dalších metod a vlastností. Projděte si je sami v [MSDN], čtěte sadu článků o vláknech v sekci „Managed Threading“.

1.3 Priority

Priority procesů a vláken v .NETu fungují stejně jako ve Windows, jde o věci přímo spojené s operačním systémem, nad kterým .NET provozujeme. U každého procesu tedy lze nastavit třídu priority, která určuje základní prioritu všech vláken. Třídu priority nastavíme pomocí property `PriorityClass`. Hodnotu lze nastavit jen na jednu z několika pojmenovaných konstant, každé třídě v operačním systému odpovídá nějaká hodnota v intervalu 0–31, tu však přímo neznáme.

Jednotlivým vláknům se priorita nastavuje pomocí property `Priority` ve třídě `Thread`. Priorita vlákna se nastavuje vždy relativně vzhledem k prioritě procesu. Opět je na výběr jen z několika pojmenovaných konstant a opět každé z nich odpovídá číslo, tentokrát může být i záporné. Způsob, jakým operační systém používá tyto hodnoty, je vysvětlen v publikaci [Kep07], kapitola Správa procesoru v praxi.

Shrnutí

V této kapitole jsme se věnovali základním prostředkům systémového programování: procesům a vláknům. V úvodu kapitoly jsme si představili třídu Process a strukturu ProcessStartInfo nesoucí spouštěcí informace procesu. Důležitou vlastností této třídy je, že to není přímo proces v operačním systému, ale jen nositel informací o procesu v jednom daném okamžiku. Stejně tak třída vlákna Thread není skutečným vláknem operačního systému, ale jen třídou nesoucí informace o vlákně v jednom daném okamžiku. Tím okamžikem je obvykle vytvoření daného objektu, v případě potřeby je ale možno nechat data objektů aktualizovat.

Vytvářením nových procesů a vláken můžeme počítač nechat provádět více operací současně. Přitom ale zvláště u vláken, které společně sdílejí paměť, musíme ještě řešit jejich synchronizaci, aby souběh neskončil výpočetními chybami či dokonce deadlockem. Tomuto tématu se budeme věnovat v několika dalších kapitolách.

Pojmy k zapamatování

- Proces
- Parametr spouštěného procesu
- Shell execute
- Přesměrování standardního vstupu a výstupu
- Vlákno
- Programový zásobník
- Stav vlákna
- Lokální uložení vlákna (thread local storage)
- Priorita

Kontrolní otázky

1. *Jak ve Windows vznikne nový proces?*
2. *Popište, jaký má při zakládání nového procesu účel tzv. process start info.*
3. *Vysvětlete pojem shell execute.*
4. *Vysvětlete, co je to přesměrování standardního vstupu či výstupu.*
5. *Proč stavy vlákna v .NETu nekorespondují úplně se stavy vlákna v operačním systému?*
6. *Co je to thread local storage (TLS)?*

2 Aplikační domény

Studijní cíle: Tato kapitola seznámí čtenáře s pojmem aplikační doména. Jde o prvek rozšiřující možnosti vláken a procesů, který přímo v operačním systému nemá ekvivalent. Kapitola je určena pouze pro pokročilé studenty.

Klíčová slova: aplikační doména

Potřebný čas: 60 minut (plus čas k vypracování úloh)

Platforma .NET používá kromě pojmů vlákno a proces ještě třetí důležitý prvek zvaný aplikační doména. Aplikační doména stojí někde mezi vláknem a procesem a přímo v operačním systému se nijak neprojevuje, je to tedy čistě prvek .NETu.

Doménu si můžeme představit jako jakýsi podproces. Doména je vždy součástí nějakého procesu a má vlastní řízenou haldu pro řízené objekty. Objekty v nějaké doméně tedy nejsou vidět v doméně jiné. To platí doslova, takže například statické součásti jsou v každé doméně znova, stejně tak statické konstruktory jsou volány v každé doméně znovu. Z hlediska operačního systému je však vidět jen proces jako celek a všechny aplikační domény v jednom procesu tedy logicky sdílejí společný adresový prostor. Tato „dvojakost“ je možná jen díky tomu, že .NET má automatickou správu paměti a přímo s pojmem „adresa“ se v něm vůbec nepracuje. Ačkoliv tedy dvě aplikační domény ve skutečnosti mají (fyzicky stejný) společný adresový prostor, programy v nich běžící to za normálních okolností nemají jak zjistit.

Aplikační doména je jednotkou izolace uvnitř procesu. Umožňuje nám tedy od sebe izolovat jednotlivé části procesu, což může být někdy výhodné a přímo v operačním systému to není možné udělat.

2.1 Třída AppDomain

Aplikační doménu reprezentuje třída `System.AppDomain`. Na rozdíl od procesů a vláken, tato třída reprezentuje přímo aplikační doménu, ne jen její obraz. Každé vlákno běžící v .NETu patří do nějaké domény, doménu aktuálního vlákna (tedy tu „svou“) získáme pomocí statické property `CurrentDomain`.

Průvodce studiem

Vidíme, že třídy `Thread`*,* `Process` *a* `AppDomain` *jsou každá v jiném prostoru jmen. Konkrétně* `AppDomain` *je přímo v prostoru jmen* `System`*, jde tedy je jednu ze základních součástí systému .NET. Naproti tomu* `Thread` *a* `Process` *jsou třídy poskytující především obraz objektů operačního systému.*

2.2 Vytvoření a zrušení domény

Novou doménu vytvoříme voláním statické metody `CreateDomain()`, jako parametr zadáme (námi zvolené) jméno domény. Pro další možnosti viz [MSDN]. Existující doménu zrušíme voláním statické metody `AppDomain.Unload()`, odstraníme tím také všechna vlákna v doméně. Zrušení domény je v .NETu jedinou možností, jak se zbavit kódu, který je již v paměti. Rušení domény je poměrně složitý proces, proto si jej popišme krok po kroku:

1. Jsou zastavena všechna vlákna se zásobníkem v rušené aplikační doméně. Vysvětlení: Vlákno v daném okamžiku „je“ v nějaké doméně, případně může být i v neřízeném kódu. Vlákna mohou při volání cizího kódu tzv. cestovat mezi doménami, při takovém přechodu však zůstává v předchozí doméně zásobník vlákna a po návratu z volání vlákno pokračuje ve vykonávání kódu (tj. funguje to stejně jako při každém jiném volání). Vlákna, která aktuálně sice jsou v jiné doméně, ale dostala se tam voláním z rušené domény (i tranzitivně, tedy nepřímo), jsou pozastavena také, protože po zrušení domény by se neměla kam vrátit.
2. Jednotlivá zastavená vlákna jsou zrušena. Zrušení vlákna bylo popsáno v předchozí kapitole, ve vlákně se tedy objeví výjimka `ThreadAbortException`. Je přitom věcí nastavení systému, zda jsou při ukončování vlákna volány mezilehlé bloky `finally`, nebo zda je toto vynecháno.
3. Je aktivována událost `AppDomain.Unload`. Jde o statickou událost, takže se k ní lze registrovat přímo na třídě `AppDomain` a handlery jsou společné pro všechny domény.
4. Garbage collector pracuje a volá všechny finalizery nedosažitelných objektů. Pokud kód finalizeru běží příliš dlouho, je násilně odstřelen.
5. Garbage collector uklidí všechnu paměť, která zbyla po doméně.

Průvodce studiem

Největší přínos aplikačních domén je právě v možnosti jejich odstranění (`Unload()`*) z paměti. Zatímco v nativním kódu ve Windows lze každý DLL soubor, který dodatečně připojíme k běžícímu procesu, také odstranit, v .NETu jednou načtené součásti procesů z paměti odstranit nelze. Jedinou výjimkou jsou právě aplikační domény, musíme však dávat pozor, abychom při práci s doménou nezavlékli část jejího kódu i do jiné domény.*

K „neštěstí“ stačí i jen odkázat se na typ v cizí doméně; abychom mohli s typem pracovat, musíme mít seskupení, které jej obsahuje, ve své doméně, takže pak už se daného DLL souboru nezbavíme.

Rušení domén v .NETu 2.0 zajišťuje zvláštní systémové vlákno (tedy neprovádí to přímo volající vlákno, ani žádné z vláken v rušené doméně). Pokud v kroku 2 není možno některé vlákno zrušit (například z důvodu vykonávání neřízeného kódu), po nějaké době čekání je rušení domény přerušeno a není dokončeno. Operace zrušení domény tedy obecně není zaručena. (Doména už by nebyla zrušena, ani kdyby někdy v budoucnu po takové události problémové vlákno přeci jen skončilo. Někdo musí znovu zavolat `AppDomain.Unload()`.)

Rušení domén se týkají dvě výjimky: V případě neúspěšného rušení domény je vyhozena výjimka `CannotUnloadAppDomainException` (na vlákně volajícím `Unload()`), v případě používání objektu již zrušené domény je vyhozena výjimka `AppDomainUnloadedException`.

2.3 Načtení kódu do domény

Doména má smysl, teprve když v ní spustíme nějaký kód. Vždy platí, že vlákno při volání kódu mezi doménami cestuje, tj. totéž vlákno může vykonávat kód střídavě v různých doménách. Samotný kód je však v každé doméně samostatný.

Metody `ExecuteAssembly()` a `ExecuteAssemblyByName()` spustí v doméně program typu exe (přesněji spustitelné seskupení). Tímto způsobem lze spouštět víc programů v jednom procesu.

Průvodce studiem

Umístěním více programů do jednoho procesu ušetříte systémové zdroje a start programu bude také rychlejší. Izolace mezi aplikačními doménami však není tak velká, jako izolace mezi procesy. Zatímco procesy se navzájem prakticky neovlivňují, domény v rámci jednoho procesu sdílejí všechny systémové zdroje, například otevřené soubory nebo jakékoliv jiné systémové handly. Špatně napsaný program umístěný do jiné domény tedy může poškodit kód v ostatních doménách stejného procesu. To se týká samozřejmě především neřízeného kódu, kde je riziko chyb obecně vyšší.

Další možností, jak připojit seskupení do domény, je voláním některé ze sady metod `Assembly.Load*()`. Tyto metody se postarají o načtení seskupení a jeho připojení do domény, nespouštějí však žádný konkrétní kód.

Voláním `GetAssemblies()` je možno zjistit, která seskupení jsou právě načtená v dané aplikační doméně. Podobnou funkci má i metoda `GetReflectionOnlyAssemblies()` vracející seznam seskupení načtených v kontextu reflexe. (Jde o seskupení načtených jen pro reflexi, jejichž kód v paměti není. Vysvětlení najdete [MSDN].)

2.4 Volání kódu mezi doménami

K obecnému volání kódu ve vzdálené doméně slouží metoda `DoCallBack(CrossAppDomainDelegate), kde CrossAppDomainDelegate` je delegát, který ani nemá parametry, ani nic nevrací. Objekt, na který se tento delegát odkazuje, by měl být serializovatelného či hodnotového typu, aby jej bylo možno transferovat do vzdálené domény. Izolace mezi doménami je v .NETu totiž rovnocenná s jakýmkoliv vzdáleným vztahem, takže použití v `DoCallback()` vlastně způsobí to, že objekt předaný formou delegátu se vlastně serializuje (převede na posloupnost bajtů), přenese se do vzdálené domény a tam se deserializuje (z posloupnosti bajtů se poskládá klon původního objektu). Z uvedeného mj. vyplývá, že při volání mezi doménami přes `DoCallback()` se ve vzdálené doméně vždy vytváří kopie dotčených objektů a originální objekty nejsou měněny.

Při volání pomocí `DoCallback()` lze technicky vzato použít také třídy zděděné z `MarshalByRefObject`, v tom případě však půjde o volání odkazem, čili povede zpět do původní domény. V konečném důsledku tedy zavoláme sami svoji doménu, což je nelogické, proto hovoříme, že jde pouze o možnost „technicky vzato“. Zajímavé příklady použití `DoCallBack()` najdete v [MSDN].

2.5 Systémové události v doménách

V aplikačních doménách je možno předejít některým systémovým chybám tím, že se pokusíme chybový stav nějak „opravit“. Není to obecný nástroj k řešení problémů, týká se skutečně jen několika specifických chyb, které mohou nastat v aplikačních doméně. Pomocí dosazení vhodného handleru lze chybu „opravit“ a předejít tak poslání výjimky do aplikace.

- Událost `AssemblyResolve` je aktivována, když se nepodaří najít seskupení. Handler tedy musí dodat chybějící seskupení.
- Událost `ReflectionOnlyAssemblyResolve` je totéž pro seskupení načítané v reflection–only kontextu.

- Událost `ResourceResolve` je vyhozena, když se nepodaří najít resource (ikonu, obrázek, atp.). Resource jsou přilinkovány do seskupení, handler tedy musí dodat chybějící resource z jiného zdroje.
- Událost `TypeResolve` je aktivována, když se nepodaří najít typ (třídu, rozhraní atp.). Handler tedy musí najít typ jinde.
- Událost `UnhandledException` je aktivována při neošetřené výjimce. Handler je tedy zavolán těsně předtím, než neošetřená výjimka způsobí ukončení aplikace.

Podobným způsobem lze sledovat další (nechybové) systémové události:

- Událost `AssemblyLoad` nastává po načtení nového seskupení do domény. Můžete díky ní například průběžně sledovat, která seskupení se načítají.
- Událost `DomainUnload` nastává při ukončování domény, handler přitom může být vyvolán na jiném vlákně než tom, které doménu nechalo ukončit. Tato událost se netýká výchozí aplikační domény procesu.
- Událost `ProcessExit` nastává při ukončování procesu. Jde o podobnou událost jako `DomainUnload`, handlery jsou tentokrát volány při ukončení procesu. Týká se všech domén v procesu, ale celkový čas ukončování je limitován na přibližně 3 sekundy. (Jde o časové omezení, které platí pro všechny typy ukončování procesů. Jeho smyslem je zajistit, aby ukončovaný proces nemohl namísto skončení zůstat například v nějaké nekonečné smyčce.)

Úloha 5. Pluginy na bázi domén

Využití aplikačních domén si vyzkoušíme na modelovém příkladu pluginů: Tedy předpokládejme, že v programu chceme mít tzv. „pluginy“. To jsou komponenty (části programu), které mají jednotné rozhraní, ale mohou být vytvářeny různými autory. S pomocí aplikačních domén docílíme toho, že hlavní program můžeme celý vytvořit dříve, než budou jiní autoři vytvářet pluginy. Pluginy budou samostatná seskupení – komponenty, které pak hlavní program použije, aniž by o nich něco předem věděl.

Postup:

1. Vytvoříte aplikační doménu pomocí `AppDomain.CreateDomain(jméno)`.
2. Nastavíte doméně oprávnění. Toto je nepovinné, omezením oprávnění však můžete pomoci bezpečnosti kódu. Nikdy si totiž nemůžeme být jisti, kdo a co v kódu pluginu vlastně napsal. Čili v praxi je vhodné a doporučené doméně oprávnění omezit, ale technicky vzato to není nutné.
3. Vzdáleně vytvoříte objekt pluginu (vzdáleně = ve vaší nové doméně) voláním `doména.CreateInstanceAndUnwrap(seskupení,typ)`. Metoda vrací zástupce (proxy) objektu. Třída pluginu by měla dědit z `MarshalByRefObject`, jinak hrozí, že si objekty pluginů zavlečete do hlavní domény a plugin by již nešel odstranit.
4. Objekt teď můžete přímo volat pomocí zástupce. Dědění `MarshalByRefObject` zajistí transparentní proxy, právě proto tedy máte zástupce objektu ve vzdálené doméně a ne přímo vytvořený objekt.

Přínosem a důvodem celé této anabáze je, že nyní máte každý plugin v jiné doméně a můžete toho využít pro odstranění pluginu z paměti bez ukončení procesu. K vyzkoušení, že vám program dobře funguje, udělejte následující test:

1. Definujte rozhraní, které budou pluginy implementovat. Bude v něm jediná operace, která vrátí string.
2. Vytvořte dva pluginy, každý vrátí jiný text ve stringu. Umístěte jeden z nich do adresáře k hlavnímu programu.

3. Hlavní program při spuštění načte plugin do samostatné domény a zavolá jeho metodu. Vypíše vrácený text a rovnou zruší doménu. Pak čeká na stisk klávesy a celý postup opakuje.
4. Spusťte hlavní program a během čekání na klávesu nahraďte DLL soubor s pluginem vaším druhým pluginem. Je-li vše v pořádku, hlavní program po stisknutí klávesy vypíše text ze druhého pluginu.

Poznámka: Pokud vám nejde odstranit DLL soubor za chodu programu, máte v programu chybu a tento plugin je stále v paměti běžícího procesu.

2.6 Izolace mezi doménami

Izolace mezi doménami je na podobné úrovni jako izolace mezi procesy. Statické proměnné jsou v každé doméně znovu, stejně tak statické konstruktory se volají v každé doméně znovu. Domény spolu nikdy nesdílejí proměnné (žádná data), s pomocí serializace však lze objekty transparentně přenášet (mají-li atribut `[Serializable]`) či volat přes proxy (jsou-li zděděné ze třídy `MarshalByRefObject`). Pro lepší pochopení této problematicky je nutno nastudovat téma serializace a .NET Remoting, viz např. [Duf06].

Z hlediska operačního systému však všechny domény v procesy vystupují jako jeden celek. Systémové prostředky jsou tedy vždy sdílené mezi doménami v rámci jednoho procesu, což znamená jednak snazší práci, pokud prostředky sdílet chceme, ale také jisté nebezpečí zavlečení chyby z jedné domény do ostatních domén v procesu. Používáme-li však jen řízené věci .NETu, což je obvyklá situace, izolace domén je dostatečná (dostatečně bezpečná).

Shrnutí

Aplikační doména je specifický prvek .NETu, který přímo v operačním systému neexistuje. Domény principiálně stojí někde uprostřed cesty mezi procesy a vlákny a jsou v .NETu důležitým prvkem, proto jsme se jim i my věnovali. Na druhou stranu jde o téma velmi pokročilé a složité, proto je vyčleněno do samostatné kapitoly, která se v základním kurzu se vynechává.

Pojmy k zapamatování

- Aplikační doména
- Systémová událost v doméně
- Plugin
- Izolace mezi doménami

Kontrolní otázky

1. *Vysvětlete, co je to aplikační doména. Proč je to prvek .NETu, a ne operačního systému?*
2. *Jaký spatřujete hlavní smysl (či přínos) aplikačních domén?*
3. *Proč je volání kódu mezi doménami tak složitá operace?*
4. *Jak mohou z použití aplikačních domén těžit pluginy?*

3 Synchronizace vláken a procesů

Studijní cíle: V předchozích kapitolách jsme se seznámili s vlákny, aplikačními doménami a procesy. Nyní přejdeme k tématu synchronizace a naučíme se používat synchronizační objekty .NETu k tomu, aby při souběhu a spolupráci vláken či procesů v programech nedocházelo k chybám.

Klíčová slova: monitor, zámek, mutex, semafor, událost, čtenáři a písaři, interlocked operace

Potřebný čas: 150 minut (plus čas k vypracování úloh)

3.1 Přehled

V této kapitole se budeme zabývat synchronizačními primitivy v .NETu. Základním z nich je monitor, což je čistě objektový synchronizační prvek, který obvykle používáme jako obyčejný zámek, ale ukážeme si i další možnosti.

Dále můžeme použít mutex (třída `Mutex`) pro výhradní přístup prostředku na bázi vláken, semafor (třída `Semaphore`) pro evidenci prostředků, jejichž počet je omezen či některý z řady signálů pro synchronizaci vláken tam, kde potřebujeme, aby různá vlákna vykonala společný kód v určitém pořadí. Ukážeme si také, že většinu synchronizačních primitiv lze používat i pro synchronizaci mezi procesy.

Dále se zastavíme i u několika dalších témat, například se naučíme používat třídu `ReaderWriterLockSlim` pro zvláštní druh zámku umožňující současné čtení více vláknům, ale zápis jen jednomu. V následující kapitole pak na zde probraná témata navážeme a probereme některé běžné metody asynchronního programování.

3.2 Monitor a zámek

Na rozdíl od všech klasických operačních systémů, objektově orientované systémy (jako .NET) používají jako základní synchronizační prvek monitor – primitivum, které bychom v klasickém operačním systému hledali marně.

Technicky se monitor podobá mutexu či kritické sekci, jelikož poskytuje především možnost provést vzájemné vyloučení pomocí tzv. „zámku“ vztaženého na část kódu. Ideologie je však odlišná, protože funkce monitoru je definována vzhledem k datům, se kterými kód pracuje, nikoliv vzhledem k samotnému kódu. Základním postulátem je: „Každý objekt má monitor.“ Při práci s monitorem tedy odpadá první starost: Vytváření a rušení synchronizačních objektů. Monitor jednoduše „je“ u každého objektu, stačí ho jen použít.

Úloha 6. Nesynchronizované počítadlo

Na úvod si musíte sami vyzkoušet, co se děje, když vlákna nejsou synchronizovaná. V tom případě může program dávat nekorektní výsledky; ukážeme si to na současném přístupu dvou či více vláken do stejné proměnné.

Nejdříve vytvořte statickou třídu Počítadlo – bude obsahovat číslo typu short (16bit integer).

```
static class Počítadlo {
  static short hodnota = 0;
  static public short ČtiHodnotu() { return hodnota; }
  static public void NastavHodnotu(short h) { hodnota = h; }
```

```
}
```

Typ **short**, který je zde použit, má 65536 možných hodnot. Pokud tedy 65536× přičteme jedničku, máme opět původní číslo.

```
for(int j = 0; j < 65536; j++) {
  short a = Počítadlo.ČtiHodnotu();
  a++;
  Počítadlo.NastavHodnotu(a);
}
```

Budete-li však tento kód provádět současně ve dvou vláknech, mělo by se počítadlo přetočit dvakrát a opět být vynulované. Ve skutečnosti však dosáhnete chybového stavu, výsledkem totiž nebude nula. Vyzkoušejte!

Ještě poznámka: Program se chová jinak na jednojádrových a vícejádrových procesorech. Pro odstranění těchto rozdílů je vhodné celou výše uvedenou smyčku v každém vlákně vícekrát opakovat. Je to totiž tak rychlé, že 65536 přičtení se stihne za jeden spin vlákna (tj. v rámci daného časového kvanta, bez přerušení jiným vláknem). Na vícejádrových procesorech počítadlo i tak bude mít chybnou hodnotu, ale abyste dosáhli viditelné chyby i na jednojádrových procesorech, udělejte to třeba takto:

```
for(int i = 0; i < 1000; i++) {
  for(int j = 0; j < 65536; j++) {
    short a = Počítadlo.ČtiHodnotu();
    a++;
    Počítadlo.NastavHodnotu(a);
  }
}
```

Červené číslo 1000 si nastavte tak velké, abyste opravdu viděli nenulový výsledek. (Při malých číslech k chybě počítadla nedojde, při moc velkých je program pomalý.)

Shrnutí úkolu

Vaším úkolem tedy je vytvořit dvě stejná vlákna. Každé bude přičítat číslo v počítadle o jedničku tolikrát, aby výsledkem byla opět nula. Ale kvůli špatné (přesněji žádné) synchronizaci vláken to nula nebude. Po skončení vláken (čekejte na ně pomocí **Thread.Join()**) vypište číslo z počítadla na obrazovku – je-li vše dle zadání, nebude to nula.

3.2.1 Zámek pro vzájemné vyloučení

Jednoduchou formu zámku pro vzájemné vyloučení získáme voláním statických metod **Enter()** a **Exit()**, je přitom potřeba zajistit, aby se **Exit()** volalo i při chybě, proto se musí použít konstrukce **try–finally**:

```
Monitor.Enter(objekt);
try { /* hlídaný kód */ }
finally Monitor.Exit(objekt);
```

V jazyce C# lze úplně stejný zámek realizovat také pomocí příkazu **lock** takto:

```
lock(objekt) { /* hlídaný kód */ }
```

V jazyce Visual Basic se totéž napíše takto:

```
SyncLock objekt
  ... kód ...
End SyncLock
```

Poznámka: Příkazy **lock** a **SyncLock** jsou jen tzv. syntaktický cukr a skutečně pouze volají statické metody **Enter()** a **Exit()** výše uvedeným způsobem.

Alternativou k `Enter()` je metoda `TryEnter()`, která umožňuje zadat časový limit čekání. Tato metoda vrací true, když dojde k získání monitoru (zamknutí) v daném časovém intervalu, nebo false, když je monitor obsazený jiným vláknem.

Monitor je reentrantní, tzn. při opakovaném volání `Enter()` či `TryEnter()` stejným vláknem je volání vždy úspěšné, protože lze libovolněkrát znovu zamknout již zamknutý zámek. Funguje to navíc zcela intuitivně: Na kolik západů zamkneme, na tolik západů pak musíme odemknout, než je zámek volný. Této vlastnosti se používá při rekurzivním volání, kdy v každé úrovni rekurze znovu objekt zamkneme a na konci volání v každé úrovni rekurze opět odemkneme. Objekt je přitom skutečně odemčen až po posledním volání v nejvyšší úrovni rekurze. (Zamykání má tedy počítadlo „západů." Rozsah počítadla je dostatečně velký, abychom jej mohli považovat za prakticky nekonečný.)

Jak správně používat zámek

Zámek se v praxi používá tak, že jako objekt určený k zamknutí uvedeme sami sebe, tedy obvykle `this`. U statických tříd není „this", ale můžeme použít objekt typu. Ve statické třídě S tedy používáme monitor objektu `typeof(S)`.

Průvodce studiem

Vysvětlení objektu typu: V objektově orientovaném prostředí .NET platí postulát: „Všechno je objekt." Ačkoliv statická třída nemá instance, takže není žádné „this", jehož monitor bychom pro zamykání mohli použít, i samotná třída či obecně každý (datový) typ je objektem. Tomuto speciálnímu objektu říkáme „objekt typu". Objekt třídy S získáme v jazyce C# pomocí operátoru `typeof`*, v jazyce Visual Basic operátorem* `GetType`*.*

Zámek na bázi monitoru se použije ve všech veřejných (public) metodách, které pracují s daty náchylnými na souběh vláken. V předchozí ukázkové úloze je pouze jedna taková „problémová" proměnná (ve třídě Počítadlo), situaci však komplikuje to, že vlákna volají střídavě ČtiHodnotu() a NastavPočítadlo(). Řešení je vaší následující úlohou.

Úloha 7. Synchronizované počítadlo

Je jasné, co vás čeká: Doplňte kód z předchozí úlohy tak, aby u počítadla nedocházelo k paměťové anomálii. Ale pozor! Nebude to tak snadné. Vy totiž musíte uzavřít čtení i zápis počítadla do jednoho bloku `lock` – zamykáním každé ze dvou metod třídy Počítadlo totiž nestačí. Proto si přidejte třetí metodu void Přičti(). Kód této metody uzamknete.

Průvodce studiem

Možná se ptáte, k čemu je vlastně dobrý zámek, když ani v jednoduchém příkladě s počítadlem nestačí jeho prostá aplikace na dvě metody. Problém zde však není v zámku, ale v oné třídě Počítadlo. Původní podoba třídy obsahující metodu NastavPočítadlo() je totiž z principu chybná, jelikož se snaží vnutit do počítadla stav zvnějšku, který je navíc v programu vypočítáván úpravou staršího stavu počítadla. Dochází tedy k tomu, že vyjmeme z počítadla jeho stav, pracujeme s ním v jiném objektu a pak počítadlu vnutíme stav nový. Tento algoritmus odporuje „dobrým mravům" v objektově orientovaném programování, z principu počítadla by tam měla být pouze jedna veřejná metoda ke změně stavu, a to právě Přičti(). Druhá veřejná metoda ČtiHodnotu() může sloužit jako

pomůcka pro zjištění aktuálního stavu, zde není žádná vada. Metoda NastavPočítadlo() by vůbec k dispozici být neměla. Rada tedy zní: „Třída nemá nikdy mít veřejné metody NastavNěco()." a může se vám hodit i v jiných situacích. (Výjimkou je samozřejmě inicializace objektu.)

3.3 Mutex

Mutex v .NETu představuje třída `System.Threading.Mutex`. Na rozdíl od monitoru, zde se jedná přímo o klasický synchronizační objekt operačního systému, jeho význam tedy známe z přednášek. (.NET nerozlišuje pojmy mutex a kritická sekce.)

V konstruktoru mutexu lze nastavit, zda má být mutex vytvořen již jako vlastněný vytvářejícím vláknem, či nikoliv. Nepovinně lze uvést také jméno mutexu; pojmenované mutexy lze použít pro synchronizaci mezi procesy.

Průvodce studiem

Koncept pojmenovaných synchronizačních objektů je ve Windows hojně rozšířen. Zatímco při práci s vlákny mají všechna vlákna přístup ke stejnému synchronizačnímu objektu prostřednictvím sdílené paměti, při práci s procesy žádnou sdílenou paměť nemáme, takže jméno slouží místo běžné objektové reference k nalezení synchronizačního objektu v dalších procesech.

Pro získání mutexu (vstup do kritické sekce) voláme `WaitOne()`, pro uvolnění mutexu (opuštění kritické sekce) voláme `ReleaseMutex()`. Vytvoření tzv. vlastněného mutexu odpovídá atomickému volání WaitOne() přímo v konstruktoru. Žádné další metody nejsou třeba.

Úloha 8. Neinstanciovatelný program

Za normálních okolností můžeme program spustit opakovaně, prakticky kolikrát chceme současně. Někdy se však může hodit omezit spouštění programu jen na jednu instanci. Toho lze dosáhnout právě pomocí mutexu. Vaším úkolem je vyzkoušet si to v praxi.

Vytvořte program, který se na začátku pokusí vytvořit pojmenovaný mutex, díky čemuž pozná, zda jde o první spuštění, či nikoliv. Program vypíše na obrazovku, zda jde o první spuštění, či nikoliv. Pokud to bude první spuštění, bude čekat na stisk klávesy (např. voláním `Console.ReadLine()` jednoduše čekáme na Enter). Pokud to první spuštění nebude, program hned skončí.

3.4 Princip čekatelných objektů

Metoda `WaitOne()` je v mutexu zděděná z předka, třídy `System.Threading.WaitHandle`. Mutex není jediným objektem typu `WaitHandle`, další jsou `EventWaitHandle` (událost) a `Semaphore` (semafor), které si představíme dále v textu. Znamená to tedy, že různé typy synchronizačních objektů lze používat stejným způsobem tak, že na ně „čekáme". Třída `WaitHandle` umožňuje čekat také na několik čekatelných objektů současně. Stačí je umístit do pole a volat statickou metodu `WaitHandle.WaitAny()` pro čekání na libovolný objekt, nebo `WaitHandle.WaitAll()` pro čekání na všechny objekty v poli.

Důležitým pomocníkem je statická metoda `WaitHandle.SignalAndWait()`, která atomicky signalizuje jeden čekatelný objekt a čeká na jiný. V operačním systému Windows je toto jediná možnost, jak atomicky signalizovat objekt a čekat na jiný a je to velmi užitečná pomůcka zvláště na víceprocesorových systémech. V prostředí .NETu lze tuto metodu nahradit signalizačními metodami monitoru, viz sekci 3.7.

3.5 Semafor

Semafor je v .NETu implementován ve třídě `System.Threading.Semaphore`, jde opět o čekatelný objekt. V konstruktoru nastavíme celkový počet zdrojů a výchozí počet volných zdrojů. Nepovinně lze uvést jméno semaforu, pojmenované semafory jsou sdílené mezi všemi procesy v systému. Uvedením jména již existujícího semaforu je namísto vytvoření nového získán odkaz na existující systémový semafor, parametry uvádějící počty zdrojů jsou pak ignorovány. Další nepovinný parametr je výstupní a signalizuje, zda byl pojmenovaný semafor nově vytvořen, či získán ze systému.

Na rozdíl od mutexu, zdroje hlídané semaforem nejsou vázány na konkrétní vlákna, takže vrátit zdroj do semaforu může jiné vlákno, než které jej získalo. Získání zdroje (vstup do semaforu) zajišťují čekací funkce (viz předchozí sekce), uvolnění zdroje zajišťuje metoda `Release()`, kde lze nepovinně uvést počet uvolněných zdrojů (standardně se uvolňuje jeden).

3.6 Vláknová afinita (thread affinity)

Vláknová afinita je pojem, který popisuje zásadní odlišnost mezi mutexem a semaforem: Mutex musíme vrátit stejným vláknem, kterým jsme jej získali. Naopak počítadlo semaforu můžeme zvyšovat a snižovat libovolnými vlákny bez omezení.

Co to znamená v praxi? Získáme-li mutex, pak jej „vlastní“ jedno konkrétní vlákno, které jej získalo. Výhodou je, že toto vlákno se může znovu pokusit získat tento mutex a systém již pozná, že jde o totéž vlákno a bez čekání mu umožní toto znovuzískání. Semafor ale takovou věc nedělá: Pokusíme-li se stejným vláknem získat další jednotku z počítadla, opravdu získáme další jednotku, a ne jen znovu tutéž, kterou už máme. Máme však možnost jednotky do semaforu vracet jiným vláknem, než kterým jej ze semaforu bereme. Tuto důležitou vlastnost můžeme použít například tak, že prostřednictvím semaforu budeme provádět jakoby transport signálů z jednoho vlákna do druhého. První vlákno bude pouze zvyšovat počítadlo (formálně tedy uvolňovat semafor), druhé vlákno bude jen snižovat počítadlo (formálně tedy vstupovat do semaforu). Druhé vlákno tedy vlastně pracuje jen tak rychle, jak rychle přicházejí povely od prvního vlákna.

Úloha 9. Omezený počet instancí programu

Tato úloha je obdobou předchozího úkolu s pojmenovaným mutexem. Dejme tomu, že chceme umožnit vícenásobné spuštění aplikace, ale jen v omezeném počtu. K tomu lze použít pojmenovaný semafor. Rozdíl od úlohy s mutexem bude jen v tom, že tentokrát musíme semafor nejen vytvořit, ale také používat, protože každá běžící instance si musí vzít svou jednotku z počítadla. Dejte také pozor na to, abyste v každém případě tuto jednotku do semaforu na konci programu vrátili.

Průvodce studiem

Chceme-li limitovat počet spuštění, ale na více než jen jedno, můžeme k tomu použít pojmenovaný semafor. Nebudeme pak sledovat jen vytvoření semaforu, každá běžící instance programu si vezme jednu jednotku pro sebe.

3.7 Signál

Signál je poměrně jednoduchý nástroj pro synchronizaci a komunikaci. Jak už víme z předchozího textu, čekatelné objekty nabízejí i možnost atomické signalizace jednoho objektu a čekání na jiný. Čekatelné objekty jsou i signály a lze s nimi díky tomu řešit úlohy, kde je třeba provádět kód ve více vláknech v určitém pořadí.

Průvodce studiem

Předchozí věta je klíčová: Signály umožní řídit pořadí provádění kódu. Bez použití synchronizace se provádí kód ve všech vláknech a to jakoby současně, přičemž nevíme, jak přesně jsou jednotlivé příkazy různých vláken prokládány mezi sebou. V některých situacích potřebujeme, aby se vlákna nějakým způsobem hlídala a prováděla kód v určitém pořadí. Pomocí signálů si vlákna mohou navzájem předávat informaci, v které části kódu se právě nacházejí a díky tomu mohou postupovat společně (synchronně).

3.7.1 Třída EventWaitHandle

Třída `EventWaitHandle` implementuje signály dle operačního systému Windows. Tuto třídu dědí třída `AutoResetEvent` a `ManualResetEvent`. Jak je patrno z jejich názvů, jedná se o třídy událostí automaticky respektive manuálně resetovaných.

Obě třídy se používají stejně: Na signál čekáme pomocí čekacích funkcí a signalizujeme jej voláním metody `Set()`. `AutoResetEvent` probudí jedno čekající vlákno a automaticky se resetuje (přestane být signalizován). Nečeká-li žádné vlákno, událost je aktivní až do příchodu prvního vlákna. Čeká-li více vláken, nelze určit, v jakém pořadí budou probouzena. (Každé volání `Set()` probudí právě jedno z nich, ale nevíme, které to bude.)

`ManualResetEvent` probouzí všechna čekající vlákna a zůstává signalizován až do zavolání metody `Reset()` kterýmkoliv vláknem, během doby v signalizovaném stavu ihned probouzí i všechna nově příchozí vlákna.

Třída `EventWaitHandle` plně nahradí `AutoResetEvent` i `ManualResetEvent`, neboť typ resetování události (automatické/manuální) se nastaví v konstruktoru. Dále zde můžeme objekt pojmenovat a vytvořit tak opět globálně přístupný systémový synchronizační objekt.

3.7.2 Signalizace pomocí monitorů

Objektové monitory lze použít i pro signalizaci, místo třídy `EventWaitHandle`. Smyslem monitorové signalizace je opět zpřehlednění kódu a navázání synchronizačních objektů přímo na objekty, jichž se synchronizace týká. Opět zde totiž využíváme faktu, že každý objekt má svůj monitor, a nemusíme žádné další synchronizační objekty explicitně vytvářet a uchovávat.

Čekání i signalizaci můžeme provádět jen uvnitř chráněné sekce (zámku) monitoru. Čekáme voláním statické metody `Monitor.Wait()`, která blokuje volající vlákno a zároveň

samozřejmě na tuto dobu opustí monitor. Signalizujeme voláním statické metody `Monitor.Pulse()`. Signalizace pomocí této metody probudí jen jedno čekající vlákno. Pokud žádné vlákno nečeká, volání `Pulse()` neudělá nic. Statická metoda `Monitor.PulseAll()` je podobná, ale probudí všechna čekající vlákna.

Průvodce studiem

Vždy musí nejprve nějaké vlákno začít čekat pomocí `Monitor.Wait()`, než jej může jiné vlákno probudit voláním `Monitor.Pulse()` či `Monitor.PulseAll()`. Pokud žádné vlákno nečeká, signál se de facto ztratí. V praxi může být někdy problém zajistit, že čekací vlákno opravdu přijde jako první, tyto situace je možno řešit použitím výše uvedené třídy `AutoResetEvent` – zatímco signály monitoru nejsou uchovávány a „na nikoho nečekají“, událost realizovaná třídou `AutoResetEvent` totiž jakoby „čeká“, až si pro ni někdo přijde.

3.8 Čtenáři a písaři

Čtenáři a písaři představují modelovou situaci speciálního případu asymetrického zamykání, kde o prostředek soupeří více vláken, ale některá z nich data jen čtou a mohou tedy pracovat současně. Problém čtenářů a písařů tedy lze implementovat jak klasickým zamykáním, které jsme probrali výše, tak pomocí speciální typu zámku, který umožňuje vláknům označeným jako čtenáři současný přístup. Čtenář totiž nemění data, ke kterým přistupuje, takže nemůže ovlivnit jiné čtenáře. Tento model je v .NETu implementován ve třídách `ReaderWriterLock` a `ReaderWriterLockSlim`.

Průvodce studiem

Třídy čtenářů a písařů obsahují poměrně mnoho součástí, kterými lze program vyladit pro maximální efektivitu. My si zde však představíme jen základní operace, které pro běžnou práci se zámky bohatě postačí. Další podrobnosti najdete v [MSDN].

3.8.1 Třída ReaderWriterLockSlim (.NET 3.5)

Systém čtenářů a písařů je implementován ve třídě `System.Threading.ReaderWriterLockSlim`, která je dostupná až ve verzi .NET 3.5.

Postup práce se zámkem je velmi podobný tomu, co bylo představeno u třídy Monitor v sekci 3.1 na straně 22. Při zamknutí však musíme určit, zda jde o čtení, zápis, nebo čtení s možností pozdějšího „upgradu“ na zápis. Zamknutí pro čtení umožní dalším vláknům, aby taky četla, zatímco zamknutí pro zápis je výlučné (tedy stejné jako u Monitoru).

Objekt zámku vytvoříme konstruktorem bez parametrů. Takto vytvořený zámek nepodporuje rekurzi, tedy nelze znovu zamknout zámek již zamknutý. Toto chování zrychlí program, nicméně odchyluje se od toho, co známe od klasického Monitoru. Chceme-li umožnit rekurzi (aby se čtenáři–písaři chovali stejně jako Monitor), uvedeme jako parametr konstruktoru hodnotu `LockRecursionPolicy.SupportsRecursion`.

Zamknutí neboli vstup do hlídané sekce provedeme voláním `EnterReadLock()` – pro čtení, `EnterWriteLock()` – pro zápis, nebo `EnterUpgradeableReadLock()` – pro čtení s možností upgradu na zápis. Tyto metody nemají žádné parametry. Při chybě, jako například

špatném vnoření, tyto metody vyhodí výjimku `System.Threading.LockRecursionException`. Tyto tři metody mají ještě alternativu s předponou `Try`…, kde jako parametr uvedeme maximální čas (dobu) čekání. Při zamknutí tyto varianty vrací `true`, nepodaří-li se v daném čase získat zámek, metody se vrací zpět s hodnotou `false`. (Jde tedy o obdobu čekacích funkcí.)

Uvolnění zámku provedeme voláním `ExitReadLock()`, `ExitWriteLock()`, nebo `ExitUpgradeableLock()`. Musíme volat metodu odpovídající typu zámku, který uvolňujeme (stejný název jako při zamykání, jen na začátku je místo Enter slovo Exit). Třída `ReaderWriterLockSlim` však toleruje míchání typů při rekurzi: Máme-li rekurzivně vnořený zámek, pak můžeme různé Exit…() volat v jiném pořadí, než jsme volali `Enter…()`.

Řada property ve třídě `ReaderWriterLockSlim` nám umožňuje zjistit aktuální stav zámku, úroveň vnoření apod. Tyto hodnoty nejsou obvykle potřeba, nebo alespoň v případě správného algoritmu. Jejich seznam a popis najdete v [MSDN].

Priority čtení a psaní

Třída `ReaderWriterLockSlim` používá plánovací strategii, která upřednostňuje písaře a ještě více vlákna v upgradovatelném režimu.

Je-li zámek obsazen, další vlákna žádající o zámek se řadí do jedné ze tří front, podle typu žádaného přístupu. V okamžiku uvolnění zámku dostane přednost vlákno čekající na upgrade ze čtecího do zápisového režimu (takové může být jen jedno), pak vlákno čekající na zápis, pak vlákno čekající na upgradovatelné čtení, pak vlákna čekající na čtení. Po dalším uvolnění zámku se tento proces opět opakuje, takže například v případě příchodu nových písařů se čtenáři nemusejí dostat ke slovu nikdy. Algoritmus tedy není ve všech případech férový. (Problémy jsou však čistě teoretické, neboť ve většině reálných situací se nevyskytuje tolik písařů, aby zcela zablokovali čtenáře.)

Pro upřesnění dodejme, že systém přechodů mezi čtecím, upgradovatelným a zápisovým režimem je ve skutečnosti ještě složitější, než je zde popsáno. Například během fáze čtení jsou nově příchozí čtenáři odbavováni bez čekání, dokud jsou fronty pro upgradovatelné čtení a zápis prázdné. Jakmile ale nejsou, nově příchozí čtenáři jsou řazeni do fronty a musejí čekat. Podrobnosti najdete v [MSDN].

Další informace

Vlákno nemůže přejít ze čtecího do upgradovatelného režimu a pouze jedno vlákno může mít upgradovatelný zámek. To znamená, že jakmile získáme zámek pro čtení, můžeme jen číst. Pouze požádáme-li od začátku o upgradovatelné čtení, můžeme během držení zámku přejít do zapisovacího režimu dodatečným voláním `EnterWriteLock()`. I když upgradovatelný režim je jen pro čtení, nelze jej povolit více vláknům současně. Kdyby dvě vlákna měla tento zámek současně a obě se rozhodla přejít do zapisovacího režimu, došlo by k deadlocku.

3.8.2 Třída ReaderWriterLock

Další varianta čtenářů a písařů je implementována ve třídě `System.Threading.ReaderWriterLock`. Tato třída se používala ve starších verzích .NETu (tj. do verze 3.0). Její nevýhodou je menší efektivita (běh programu je pomalejší), což však v řadě reálných situací vůbec není problém.

Průvodce studiem

Starší třída `ReaderWriterLock` se vám bude hodit zejména tehdy, když by novější a lepší `ReaderWriterLockSlim` byl jediným důvodem vašeho přechodu na .NET 3.5.

Bude-li program vyžadovat jen například .NET 2.0, můžete se tím vyhnout problémům se spouštěním na některých neaktualizovaných počítačích, kde jednoduše nic novějšího než .NET 2.0 není. (Verze .NET 2.0 je z roku 2005, verze .NET 3.5 z roku 2007, je v okamžiku psaní tohoto textu čerstvě vydaná. Lze samozřejmě očekávat, že po nějaké době bude již .NET 3.5 tak rozšířený, jako je dnes .NET 2.0.)

Použití tohoto zámku je podobné jako u „slim" verze, metody se však jmenují jinak. Zamknutí provedeme voláním `AcquireReaderLock()` – pro čtení, nebo `AcquireWriterLock()` – pro zápis. Obě metody berou jako parametr maximální dobu čekání, hodnota –1 znamená čekat nekonečně dlouho. Funkce „upgrade" zámku ze čtení na zápis je zde podporována vždy, takže nepotřebuje zvláštní metody.

Uvolnění zámku provedeme voláním `ReleaseReaderLock()`, nebo `ReleaseWriterLock()`. Třída `ReaderWriterLock` je vždy rekurzivní, takže zamykání a odemykání vždy používá vnitřní počítadlo (čili chová se jako třída `Monitor`).

Popis dalších (pokročilejších) funkcí této třídy najdete v [MSDN].

Priority čtení a psaní

Třída `ReaderWriterLock` zajišťuje spravedlivé střídání mnoha čtenářů a jednoho písaře.

Žádá-li více vláken o zámek, tato jsou řazena do dvou front – zvlášť pro čtení a pro zápis. V okamžiku ukončení zápisu je povolen (současný) vstup všem vláknům čekajícím na čtení. Po skončení všech těchto čtení je vybráno první vlákno ze zápisové fronty a pouze jemu je vstup povolen. Čeká-li v zápisové frontě více vláken, další se dostanou ke slovu až po dalším kolečku čtenářů. Pozor však na to, že přicházejí-li další čtenáři v okamžiku, kdy probíhá čtení, těmto není umožněno čtení, nýbrž jsou zařazeni do fronty! Ke slovu se dostanou až po ukončení čtení aktuálních čtenářů a jednom zápisu (je-li někdo v zápisové frontě).

Úloha 10. Čtenáři a písaři

Vyzkoušíme si třídu `ReaderWriterLock`. K dispozici máte tuto kostru programu:

```
static class Program {
  static volatile int[] pole = new int[100];
  static bool chcekončit = false;

  static public void Písař() {
    Random r = new Random();
    while(!chcekončit) {
      lock(typeof(Program)) {
        for(int i = 0; i < pole.Length; i++) pole[i] = 0;
        Thread.Sleep(100);
        for(int i = 0; i < pole.Length; i++)
          while(pole[i] == 0) pole[i] = r.Next();
      }
      Thread.Sleep(100);
    }
  }

  static public void Čtenář () {
    int suma=0;
    while(!chcekončit) {
      lock(typeof(Program)) {
        for(int i = 0; i < pole.Length; i++) {
          suma += 10 / pole[i];
          Thread.Sleep(1);
        }
```

```
      }
      Console.Write(".");
    }
  }

  static void Main(string[] args) {
    Thread písař = new Thread(Písař);
    Thread[] čtenáři = new Thread[10];
    písař.Start();
    Thread.Sleep(500);
    for(int i = 0; i < čtenáři.Length; i++) {
      čtenáři[i] = new Thread(Čtenář);
      čtenáři[i].Start();
    }

    Console.ReadLine();
    chcekončit = true;
    písař.Join();
    for(int i = 0; i < čtenáři.Length; i++) čtenáři[i].Join();
  }
}
```

Následuje tentýž program ve Visual Basicu:

```
Module Module1
  Dim pole(100) As Integer
  Dim chcekončit As Boolean = False

  Sub Písař()
    Dim r As Random = New Random()
    While chcekončit = False
      SyncLock GetType(Module1)
        For i As Integer = 0 To pole.Length - 1
          pole(i) = 0
        Next
        Threading.Thread.Sleep(100)
        For i As Integer = 0 To pole.Length - 1
          While pole(i) = 0
            pole(i) = r.Next
          End While
        Next
      End SyncLock
      Threading.Thread.Sleep(100)
    End While
  End Sub

  Sub Čtenář()
    Dim suma As Integer = 0
    While chcekončit = False
      SyncLock GetType(Module1)
        For i As Integer = 0 To pole.Length - 1
          suma = suma + 10 / pole(i)
          Threading.Thread.Sleep(1)
        Next
      End SyncLock
      Console.Write(".")
    End While
  End Sub

  Sub Main()
    Dim _písař As Threading.Thread = New Threading.Thread(AddressOf Písař)
```

```
    Dim čtenáři(10) As Threading.Thread
    _písař.Start()
    Threading.Thread.Sleep(500)
    For i As Integer = 0 To čtenáři.Length - 1
      čtenáři(i) = New Threading.Thread(AddressOf Čtenář)
      čtenáři(i).Start()
    Next

    Console.ReadLine()
    chcekončit = True
    _písař.Join()
    For i As Integer = 0 To čtenáři.Length - 1
      čtenáři(i).Join()
    Next
  End Sub
End Module
```

Popis programu: Program obsahuje sdílenou proměnnou pole, dále pak metodu Písař, kterou vykonává vlákno písaře, a metodu Čtenář, kterou vykonává každé vlákno čtenáře. Písař je zde jen jeden, čtenářů je deset. Kód v těchto metodách je jen fiktivní, jeho smyslem je simulovat situaci, kdy písař během psaní uvede sdílenou proměnnou do nekonzistentního stavu a čtenáři nesmějí tuto proměnnou číst, dokud písař nezapíše nový stav a proměnná nebude opět konzistentní. (V našem příkladě je nekonzistentním stavem pole, ve kterém jsou nuly. Kód čtenáře pak padá s chybou dělení nulou.)

Výše uvedená kostra kódu je funkční, k zamykání ale používá obyčejné zámky. Program je tak velmi pomalý, protože čtenáři nemohou pracovat společně. V kódu je vidět, že písař simuluje, že jeho psaní trvá 0.1s a po dokončení čeká jen 0.1s a opět opakuje zápis. Písař tedy nechává de facto jen polovinu času procesoru všem čtenářům, druhou polovinu tráví psaním. Čtenářů je však hodně a každý by chtěl číst. V daném krátkém čase 0.1s se však všichni ke čtení vůbec nedostanou. Kód čtenáře simuluje práci s polem, kdy počítá jistou zvláštní formu sumy hodnot v poli. Přitom simuluje, že zpracování jedné položky v poli trvá 0.001s, což při našem 100prvkovém poli znamená zpracování celého pole za 0.1s (tedy stejný čas jako u písaře). Po zpracování celého pole vypíše čtenář na obrazovku tečku.

Metoda `Main` pak dává do „bojového ringu" jednoho písaře a 10 čtenářů, každý přitom chce pořád pracovat, ale kvůli zámkům může pracovat jen jeden. Vaším úkolem je použít pro zamykání třídu `ReaderWriterLock`, čímž se program o hodně zrychlí.

Průvodce studiem

Nezapomeňte, že v sekcích kódu používajících zámky čtenářů a písařů musíte použít blok try–catch tak, jak to bylo ukázáno v sekci 3.2.1 u obyčejného zámku.. Jinak totiž při výskytu chyby a výjimky program přeskočí uvolnění zámku a tím je narušena konzistence programu (a velmi vážně hrozí dreadlock).

3.9 Blokované (interlocked) operace

Blokované operace, nebo lépe přímo anglicky interlocked, slouží k synchronizaci bez zamykání. Zatímco všechny předchozí sekce pojednávaly o zamykání, protože ať už šlo o monitor, mutex, semafor či událost, vždycky tam docházelo k nějakému čekání na získání přístupu, interlocked operace se provádějí okamžitě a bez čekání.

Výhodou interlocked operací je především rychlost výsledného kódu. Prvním důvodem vyšší rychlosti je, že vlákna na sebe nikdy nemusejí čekat, neboť každá operace je provedena okamžitě. Druhou výhodou je, že díky absenci zámků je celý kód rychlejší už z toho důvodu, že právě odpadá složitá synchronizace operací ošetřující práci se zámkem.

Interlocked operace jsou jednoduché operace, které jsou provedeny atomicky bez toho, aby byla použita nějaká forma zámku (v obecném slova smyslu). Například v úloze s počítadlem jsme použili zámek k tomu, abychom provedli zvýšení počítadla o jedničku atomicky, neboť operace přičtení sama o sobě atomická není. Tuto úlohu však lze řešit i pomocí interlocked operace atomické inkrementace.

Všechny interlocked operace jsou soustředěny jako metody ve statické třídě `System.Threading.Interlocked`.

3.9.1 Základní aritmetika

`Increment()` – Atomicky zvýší hodnotu proměnné typu `int` či `long` o jedničku.

`Decrement()` – Atomicky sníží hodnotu proměnné typu `int` či `long` o jedničku.

`Add()` – Atomicky přičte k proměnné typu `int` či `long` danou hodnotu. Totéž lze použít i pro odečtení, stačí obrátit znaménko hodnoty.

Tyto základní operace změní hodnotu proměnné, fungují jen pro `int` a `long` a nevracejí původní, ani novou hodnotu měněné proměnné. Poslední základní operací je atomické přečtení 64bitové hodnoty.

`Read()` – Atomicky přečte a vrací hodnotu proměnné typu `long`. Má smysl jen na 32bitových systémech, protože čtení proměnných o počtu bitů rovném počtu bitů systému je atomické vždy. (Práce s typem `double` je také atomická, protože jde přes 64bitové registry MMU.)

3.9.2 Složitější aritmetika

`Exchange()` – Atomicky vymění hodnotu proměnné s danou konstantou a vrací původní hodnotu proměnné. (Odpovídá instrukci procesoru `xchg`, ale je provedeno atomicky.) Funguje pro různé datové typy, včetně generické varianty pro referenční typy.

`CompareExchange()` – Atomicky provede porovnání a výměnu hodnoty proměnné. Tato metoda provádí atomicky operaci CAS (definovanou v teorii operačních systémů). Jedná se o nejsložitější interlocked operaci, je však stále provedena atomicky a bez zámků. Metoda má tři parametry: Prvním je odkaz na proměnnou, jejíž hodnota je porovnána se třetím parametrem. Jsou-li rovny, pak je hodnota proměnné nahrazena druhým parametrem. Metoda navíc vrací původní hodnotu proměnné, bez ohledu na výsledek testu rovnosti. Tato metoda je opět poskytována pro různé datové typy, včetně generické varianty pro referenční typy.

Úloha 11. Interlocked počítadlo

Interlocked operace vyzkoušíme opět na úloze s počítadlem. Vaším úkolem je implementovat další variantu počítadla pomocí interlocked operace. Pochopitelně k tomu bude stačit nejjednodušší operace `Increment()`, k otestování nám to však stačí. Všimněte si také, že se program zrušením zámků zrychlí.

Shrnutí

Synchronizace vláken a procesů je jedním z hlavních témat tohoto učebního textu. Poté, co jsme se seznámili s prací s vlákny, aplikačními doménami a procesy v předchozích kapitolách, jsme se nyní pustili do studia samotných synchronizačních objektů.

V úvodu kapitoly jsme si představili třídu Monitor, který je základním synchronizačním prvkem nejen v .NETu, ale pro svou objektovou povahu i v dalších objektově orientovaných prostředích. Dále jsme si představili mutexy, semafory a signály a vysvětlili jsme si systém čekatelných objektů, který platforma .NET zdědila ze systému Windows, nad kterým je vystavěna.

V další části kapitoly jsme si představili zámky pro čtenáře a písaře, které výrazně zrychlí ty algoritmy, kde mnoho vláken soupeří o prostředek, ale jen málokteré jej mění. Tato problematika je poměrně složitá, omezili jsme se proto jen na základní funkcionalitu, která nám však při běžné práci bohatě postačuje.

V poslední části kapitoly jsme probrali tzv. interlocked, česky nepříliš srozumitelně „blokované" operace, které poskytují možnost provádět některé vybrané jednoduché operace bezpečně mezi vlákny bez použití jakýchkoliv zámků či jiného typu synchronizace. Dokážeme-li si s těmito jednoduchými interlocked operacemi vystačit, získáme maximální rychlost běhu programu, protože všechny typy zámků, mutexy, semafory atp. program zpomalují.

Pojmy k zapamatování

- Monitor
- Zámek
- Mutex
- Čekatelný objekt
- Semafor
- Vláknová afinita
- Signál
- Čtenáři a písaři
- Blokované (interlocked) operace

Kontrolní otázky

1. *Vysvětlete postulát: „Každý objekt má monitor." Jak se liší od jiných modelů?*
2. *Jak byste pomocí monitoru implementovali mutex?*
3. *Co je to čekatelný objekt (tj. co jej charakterizuje)?*
4. *Jak byste pomocí monitoru implementovali semafor?*
5. *Co je to vláknová afinita?*
6. *Diskutujte rozdíly mezi třemi druhy signálů v .NETu, které byly představeny v této kapitole.*
7. *Vysvětlete, proč mají zámky čtenářů a písařů vliv na rychlost programu.*
8. *Vysvětlete rozdíl mezi čtecím a upgradovatelným režimem zámku.*
9. *Proč má .NET dvě různé třídy implementující zámek čtenářů a písařů?*
10. *Vysvětlete, co jsou to blokované (interlocked) operace. Jaký je jejich hlavní přínos?*

4 Asynchronní výpočetní techniky

Studijní cíle: Tato kapitola je volným pokračování kapitoly předchozí. Známe už synchronizační primitiva a nyní se naučíme několik praktik běžných v asynchronním programování v rámci platformy .NET.

Klíčová slova: fond vláken, časovač, APM, background worker, roura, aktualizace GUI

Potřebný čas: 140 minut (plus čas k vypracování úloh)

4.1 Přehled

V této kapitole navazujeme na znalosti z kapitoly předchozí. Synchronizační primitiva už známe, tentokrát se seznámíme s prostředky, které pro snazší synchronizaci a komunikaci nabízí .NET. Jsou to všechno věci, které bychom si dokázali naprogramovat i sami pomocí základních konstruktů, které již známe. Jelikož však paralelní programování je věc složitá a spousta postupů se tam navíc opakuje, knihovna .NETu nabízí celou řadu již zabudovaných věcí, které nám při asynchronním programování život zjednoduší.

Některé z věcí, které jsou obsahem této kapitoly, používá i samotné prostředí .NETu, například fond vláken je častým zdrojem vláken, která tu a tam systém potřebuje k provedení nějaké krátké operace na pozadí. Vlákna ve fondu se recyklují, což šetří systémové zdroje (vytvoření a zrušení vlákna je náročná operace). Představíme si také několik verzí časovače pro přesně měření času, naučíme se používat vlákna v GUI aplikacích a toto pak zobecníme do vzoru asynchronního programování, který lze nasazovat v libovolné situaci, kde se nám vykonávání operací na pozadí může hodit. Navážeme pak ještě několika dalšími úzce souvisejícími tématy a na konci kapitoly zmíníme také roury jakožto prostředek sloužící ke komunikaci mezi procesy na lokálním počítači i síti.

4.2 Fond vláken (ThreadPool)

4.2.1 Úlohy na pozadí

V .NETu můžeme s vlákny pracovat i jinak než přímým použitím třídy `Thread`. Jednou z alternativ je tzv. fond vláken reprezentovaný třídou `System.Threading.ThreadPool`. Je to statická třída umožňující provádět úlohy na pozadí. Použití fondu vláken je jednodušší než vytváření plnohodnotných vlastních vláken, nad úlohami prováděnými takto na pozadí však nemáme žádnou kontrolu (jsou prostě v pozadí a nestaráme se o ně), proto se to nemusí hodit ve všech situacích.

Použití `ThreadPool` je velmi jednoduché: Zavoláme statickou metodu `QueueUserWorkItem()` a jako parametr uvedeme metodu, která má být spuštěna na pozadí. (Jako nepovinný parametr lze navíc uvést jeden libovolný parametr, který je předán do naší metody.) Fond vláken `ThreadPool` používá jen omezený počet vláken (standardně 25, lze to ale změnit) a pokud mu dáte víc úloh, budou tyto spouštěny/vykonávány postupně po dokončení úloh dřívějších.

Použití fondu vláken může být i rychlejší než práce s třídou Thread, protože vlákna po dokončení práce nejsou zrušena, ale zůstávají ve fondu a čekají na další práci. Rychlostní rozdíl je samozřejmě patrný zejména v situacích, kdy provádíme velký počet krátkých jednoduchých úloh.

Úloha 12. Zpožděná konzola

V této úloze si vyzkoušíte `ThreadPool`. Napište konzolovou aplikaci, která bude fungovat jako zpožděná konzola. Program bude číst klávesnici a stisknuté klávesy vypisovat na obrazovku. Každá zmáčknutá klávesa se ale vypíše na obrazovku nejprve jako tečka a až po 3 sekundách se na tom místě objeví místo tečky znak stisknuté klávesy. Klávesa Enter ukončí program.

Jak na to: Především budete potřebovat datovou strukturu, ve které budete uchovávat řádek textu a počet stisknutých nezpracovaných kláves, neboli teček. Metoda Main pak bude ve smyčce volat `Console.ReadKey(true)` a jednotlivé klávesy předávat ke zpracování vláknům na pozadí.

Program bude používat pomocnou proměnnou, string společný pro všechna vlákna. Bude si v něm průběžně uchovávat celý text, který je vidět na obrazovce, včetně teček. Metoda vlákna na pozadí nejprve přidá klávesu jako tečku na konec stringu a nechá jej vypsat celý od začátku řádku. (Kurzor na začátek řádku přesunete pomocí znaku '\r'.) Potom čeká 3 sekundy, tečku ve stringu nahradí skutečným znakem a opět celý string vypíše.

Pozor: Ačkoliv race condition je zde málo pravděpodobné, práci s textovým bufferem a vypisování na obrazovku byste měli mít v bloku `lock`.

4.2.2 Asynchronní čekání

Fond vláken lze také použít pro asynchronní čekání na libovolný čekatelný objekt (`WaitHandle`). Je to stejné, jako bychom čekali pomocí některé čekací funkce třídy `WaitHandle`, ale aktuální vlákno není blokováno. Místo toho určíme delegát, který se má zavolat po skončení čekání.

Asynchronní čekání inicializujeme pomocí statické metody `ThreadPool.RegisterWaitForSingleObject()`, parametry jsou: objekt k čekání, delegát k zavolání, libovolný parametr k předání do delegátu, timeout, příznak opakování.

Delegát je zavolán nejen při signalizaci objektu, ale také po skončení daného časového intervalu (timeoutu). Je typu `WaitOrTimerCallback` a jedním z jeho parametrů je příznak timeoutu, takže delegát ví, zda je volán při signalizaci objektu, na který se čekalo, nebo při timeoutu. Příznak opakování umožňuje po zavolání delegátu opakovat čekání.

Volání `ThreadPool.RegisterWaitForSingleObject()` vrací objekt typu `RegisteredWaitHandle`. Ten pak můžeme použít pro zrušení automaticky opakovaného asynchronního čekání, stačí na něm zavolat metodu `Unregister()`.

Průvodce studiem

Všimněte si způsobu, jak se ruší automaticky opakované asynchronní čekání. Jedná se o čistý objektový ekvivalent tzv. handlů či pointerů známých z neobjektových prostředí. Registrační funkce by v neobjektovém prostředí vracela nějaký handle (číslo či pointer) sloužící k pozdější identifikaci daného čekání. Tento handle bychom pak předali do nějaké odregistrační metody, která by byla součástí `ThreadPool`*. Objektové řešení však vrací plnohodnotný objekt a pro odregistraci již nepoužíváme* `ThreadPool`*, nýbrž voláme* `Unregister` *přímo na objektu reprezentujícím výsledek registrace.*

4.2.3 Konfigurace fondu

Fond vláken udržuje jistý počet „živých“ vláken a ten upravuje dle aktuální potřeby, tedy podle počtu zadaných úloh. Vždy se však drží v jistých definovaných mezích (minimum–maximum).

Přesný minimální a maximální počet vláken obvykle nepotřebujeme znát, ve speciálních případech se však může hodit možnost je zjistit či dokonce změnit. K tomu slouží čtyři intuitivní statické metody třídy ThreadPool: `GetMinThreads()`, `SetMinThreads()`, `GetMaxThreads()`, `SetMaxThreads()`.

Zajímavější než zjišťování a nastavování těchto hodnot, však může být znalost, jak systém funguje sám o sobě. Na starších verzích .NETu byl počet vláken ve fondu 1–25 (na proces). Jak uvádí [Duf07], od verze .NET 2.0 SP1 je maximální počet zvýšen až na 250 a rychlost vytváření nových vláken je po překročení počtu procesorů 2 za sekundu. Vláken tedy může být velmi mnoho, ale vznikají velmi pomalu (dosažení maximálního počtu trvá přes 2 minuty). Velký maximální počet vláken ve fondu řeší nejčastější problém s vlákny v .NETu, kterým je nepravidelný výskyt deadlocků v okamžicích, kdy všech 25 vláken fondu čekalo na dokončení jobů, které byly v čekací frontě fondu a nebyly ještě spuštěny. Zpomalení vytváření nových vláken pak má za cíl donutit autory nesprávných konstrukcí v kódu, aby svůj kód opravili. (Vytvářet nárazově obrovská množství jobů, když máme jen několik málo procesorů, je velmi špatný programátorský styl.)

4.3 Časovače (třikrát jinak)

Časovač je jednoduchý nástroj, který nám umožní vykonat určitý kód s daným zpožděním, tedy až po uplynutí přesně daného času, a to i opakovaně v pravidelných intervalech. Použití časovače je velmi jednoduché: Stačí určit, co se má vykonat, za jak dlouho se to má vykonat a jestli se to má takto vykonávat opakovaně. V praxi se časovače využívají především pro ono zmíněné opakované provádění něčeho, ale není to podmínkou.

Platforma .NET nabízí tři různé třídy časovače, specializované pro různé situace. Každá z těchto tříd má jiné veřejné rozhraní, takže na začátku si musíme vybrat, kterou variantu použijeme. Potom jejich použití je ale právě proto, že jsou specializované, velmi snadné. Kód určený k vykonávání časovačem budeme dále nazývat „obsluha“ (důvodem je samozřejmě úzká souvislost časovače s přerušením).

Třída `System.Timers.Timer` je (serverová) komponenta a je určená k použití na vícevláknových serverech. Jeho zvláštností je, že obsluhu časovače umí provádět na různých vláknech. Toto je však mimo tematického zaměření této publikace, proto se serverovými časovači nebudeme dále zabývat.

Třída `System.Windows.Forms.Timer` je rovněž komponenta, ale tato je určena k použití v běžných „okenních“ programech pro Windows. Nepoužívá vlákna, obsluha je volána pomocí fronty zpráv. Tato třída odpovídá objektu časovače systému Windows a má poměrně malou přesnost (uvádí se přibližně 1/18 sekundy).

Třída `System.Threading.Timer` není komponentou, spíše ji můžeme chápat jako rozšíření základní třídy vlákna. Tato třída není určena k práci s GUI, obsluha je vykonávána na vláknech z fondu vláken (viz sekci 4.1 na straně 35). Tato třída přibližně odpovídá objektu multimediálního časovače, má tedy přesnost v řádu jednotek milisekund (1 až 5 ms, dle verze Windows). Rozdíl oproti Windows je v použití fondu vláken.

Obecný postup použití časovače:

1. Vytvoříme objekt časovače.
2. Nastavíme, co se má volat jako obsluha.
3. Nastavíme rychlost opakování.
4. Spustíme časovač.
5. Jakmile časovač nepotřebujeme, zastavíme jej.

4.3.1 Třída System.Windows.Forms.Timer

Třída `System.Windows.Forms.Timer` je speciálně určena pro práci s GUI. Jelikož ve Windows je nutno práce s GUI oknem dělat vždy jen vláknem, které okno vytvořilo, tento bezvláknový časovač nám ušetří mnoho práce.

Časovač vytvoříme konstruktorem bez parametrů. Událost `Tick` nastavíme na obsluhu, obvykle zde necháme zavolat jinou metodu naší třídy okna formou `Tick += metoda`, nebo pomocí anonymního delegátu přímo vepíšeme požadovaný kód. Dále ještě nastavíme rychlost časovače pomocí property `Interval`. Časovač lze takto připravit i předem (například v konstruktoru okna). Voláním metody `Start()` se časovač spustí a běží neustále až do zavolání `Stop()`. Chceme-li tedy jen jedno zavolání obsluhy, hned na prvním řádku obsluhy časovač vypneme voláním `(sender as Timer).Stop();`.

Úloha 13. Oznámení chyby červeným bliknutím

Časovač můžeme použít pro zlepšení kvality uživatelského rozhraní, vyzkoušíme si to na jednoduchém příkladu. Vytvořte program s jednoduchým formulářem pro zadání jména. V okně bude jen nápis „Zadej jméno“, pod ním místo pro zadání jména a tlačítko OK.

Průvodce studiem

Jelikož procvičujeme okenní časovač, je nutno vytvořit okenní aplikaci. Nemáte-li zatím vůbec žádné zkušenosti s tímto typem programu, bude se vám hodit několik základních informací: Program vytvořte ve Visual Studiu 2005 jako typ „Windows Application“ nebo ve Visual Studiu 2008 jako „Windows Forms Application“. Objeví se vám formulář, kam naskládáte z toolboxu (to je lišta ovládacích prvků) prvky Label – pro nápis, TextBox – pro zadání jména a Button – tlačítko. U nápisu a tlačítka nastavte ve vlastnostech (lišta Properties) položku Text. U textboxu si nastavte položku (name), která udává jméno objektu.

Potom dvojklikem na tlačítko OK přejdete k psaní kódu, který se zavolá v okamžiku stisknutí tohoto tlačítka. Napište sem kód pro zablikání dle zadání, to už je váš úkol.

Program požádá uživatele o zadání jména a po stisknutí tlačítka OK skončí. Stiskne-li uživatel OK bez zadání jména, program neskončí a zadávací box na půl sekundy zčervená. Po půl sekundě se barva vrátí zpět.

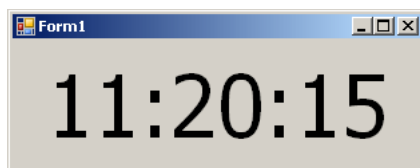
Průvodce studiem

Vyvarujte se klasické programátorské chybě: Předpokládáte-li, že standardní pozadí textboxu je bíle, je to omyl! Při otevření okna si nejprve uložte původní barvu a pak ji pouze „blikejte“ mezi červenou (`Color.Red`) a tou původní. Program tak bude fungovat ve všech situacích. Ověřte, že pole svítí červeně půl sekundy i při opakovaném stisku OK.

Všimněte si, že u této úlohy nevadí poměrně malá přesnost okenního časovače. Jde o zcela typickou úlohu pro tento časovač: Příjemcem časovacího signálu je v konečném důsledku jen člověk a ten si drobné časové odchylky v řádech zlomků sekundy nevšimne, nebo na něj přinejmenším nepůsobí rušivě. Časovač je zajímavý objekt, vyzkoušejme si jej proto ještě v další úloze.

Úloha 14. Hodiny s okenním časovačem

Vytvořte program zobrazující aktuální čas. Program bude čas (hodiny:minuty:sekundy) vypisovat do okna pomocí ovládacího prvku Label, kterému nastavíte větší font a přizpůsobíte mu i velikost okna. V okně nebude žádné tlačítko, program půjde zavřít křížkem nahoře vpravo, viz následující obrázek.



Obr. 1 Hodiny

K zobrazování hodin využijte okenní časovač, v obsluze si aktuální čas zjistíte pomocí `DateTime.Now` (je to statická položka, nepotřebujete žádné proměnné). Můžete jej také přímo naformátovat voláním `DateTime.Now.ToLongTimeString()`.

4.3.2 Třída System.Threading.Timer

Třída `System.Threading.Timer` přibližně odpovídá objektu multimedia timer systému Windows, je to tedy jakýsi nativní systémový časovač s přesností v řádu jednotek milisekund. Tento časovač používá vlákna z fondu vláken, na němž spouští obsluhu, a nedá se tedy použít ve scénářích vyžadujících aktualizaci GUI. Pozor také na to, že je-li obsluha časově náročná a jedno volání trvá déle než je čas časovače, obsluha může být volána vícekrát současně na různých vláknech z fondu.)

K dispozici je několik konstruktorů, všechny s parametry. Prvním parametrem konstruktoru je vždy obsluha ve formě delegátu `TimerCallback`.

```
delegate void TimerCallback();
```

Parametr v tomto delegátu je pro potřeby uživatele, umožňuje nám předat libovolný parametr z místa, kde voláme konstruktor, do obsluhy přerušení (sám časovač jej na nic nepoužívá).

Další z celkem čtyř parametrů konstruktoru udávají (v daném pořadí): hodnotu k předání do obsluhy, dobu do prvního zavolání obsluhy (due time), dobu mezi dalšími voláními obsluhy (period). Jedna varianta konstruktoru je jednoparametrová, ta dosadí za další parametry objekt časovače a nekonečno (`Timeout.Infinite`) za oba časy.

Rozlišení času „due“ a „period“ můžeme využít k určení, kdy má časování začít. Hodnota „period“ v posledním parametru vždy určuje rychlost časovače. Hodnotu „due“ obvykle jednoduše nastavíme na tutéž hodnotu, anebo na nulu, když chceme, aby první aktivace obsluhy proběhla ihned. V případě, že používáme časovač k jednorázovému zavolání obsluhy bez dalšího opakování, dáme do parametru „due“ požadovaný čas a parametr „period“ nastavíme na nekonečno (`Timeout.Infinite`).

> *Časy v systémovém časovači určujeme pomocí typu `TimeSpan`. Je to jeden ze základních typů .NETu sloužící právě k určení doby. Na rozdíl od typu `DateTime`, který slouží k určení doby v podobě „kdy“, typ `TimeSpan` slouží k určení doby „jak dlouho“. Hodnotu `TimeSpan` nejčastěji vytvoříme odečtením dvou `DateTime`.*
>
> *Ruční vytváření `TimeSpan` je trošku složitější, časovač i další podobné třídy však umožňují zadat čas přímo v milisekundách (hodnotou typu int). Například pro určení rychlosti 10x za sekundu stačí zadat číslo 100.*

Tento typ časovače je po vytvoření objektu okamžitě aktivní a zastaví se až voláním metody `Dispose()`. Nejsou zde tedy žádné metody ve stylu `Start()` či `Stop()`. Na živém časovači lze pouze upravovat časy voláním metody `Change(due,period)`.

Tento typ časovače si procvičíme až v další sekci, spolu s asynchronní aktualizací GUI.

4.4 Vzor asynchronní aktualizace GUI

Jak už víme z předchozí sekce, u okenních aplikací platí striktní pravidlo, že uživatelské rozhraní lze obsluhovat jen vláknem, které vytvořilo okno. Toto pravidlo platí doslova, tj. ve vícevláknových programech lze jakékoliv operace s objekty uživatelského rozhraní provádět jen vláknem, které vytvořilo okno. (Okno obvykle vytvoříme prvním vláknem při startu programu, proto se někdy používá termín „první vlákno“ pro označení vlákna obsluhujícího uživatelské rozhraní. Jindy se tomuto vláknu také říká „GUI vlákno“.)

Přímo ve Windows není spolupráce dalších vláken v okenních aplikacích nijak řešena. V .NETu je však zabudovaná elegantní pomůcka, která nám přístup ke GUI umožní. Je to metoda `Invoke()`, která je k dispozici ve všech ovládacích prvcích. Její použití spočívá v doplnění stejného kódu na začátek všech metod, které chceme umožnit volat z jiných vláken. Tento „stejný kód“ právě nazýváme vzorem asynchronní aktualizace GUI.

Postup:

1. K dané metodě vytvoříme delegát (jako pojmenovaný typ) se stejnými parametry.

```
delegate návrat delegát (parametry);
```

2. Kód metody upravíme takto:

```
návrat Metoda(parametry) {
  if(InvokeRequired) {
    return Invoke(new delegát(Metoda), new object[]{parametry});
  } else {
    ...původní kód metody...
  }
}
```

Slova *parametry* a *návrat* zde označují vstupní parametry a návratový typ naší metody. Metoda si nejprve pomocí `InvokeRequired` sama otestuje, na kterém vlákně je volána. Je-li to na prvním vlákně, provede svůj kód. V opačném případě volá metodu `Invoke()`, které předá delegát na sebe a pole s hodnotami parametrů k dosazení při volání na prvním vlákně. (Tyto hodnoty musíte zadat stejné, jako jsou při aktuální volání.) Tím je naše práce hotova.

Systém v metodě `Invoke()` předá delegát metody a hodnoty parametrů prvnímu vláknu přes frontu zpráv. Naše metoda je pak zavolána na prvním vláknu, ovšem může to chviličku trvat, pokud ve frontě ještě čekají i jiné zprávy. Vrací-li metoda něco, je tato hodnota předána zpět volajícímu vláknu do metody `Invoke()`. Stejně tak veškeré výjimky jsou předány zpět do volajícího vlákna. (Toto je velmi důležitá vlastnost!) Volající vlákno mezitím čeká, až první vlákno skončí s voláním metody a teprve potom se vrací zpět z `Invoke()`, načež dle výše uvedeného vzoru je ukončena i naše metoda.

Poznámky:

- Ke každé metodě musíme vytvořit pojmenovaný typ delegátu. Jelikož metoda `Invoke()` přijímá jakýkoliv typ delegátu, nestačí v tomto případě zadat při volání jen jméno své metody. (Toto se týká aktuální verze C#, v budoucnu to může být jinak.)
- Metoda `Invoke()` je typu `object`. Pokud naše metoda volaná v prvním vláknu něco vrací, Invoke vrací tuto hodnotu volajícímu vláknu (stačí ji pak přetypovat na správný

typ). U metod, které nic nevracejí, vynecháme příkaz `return` (ve vzoru u volání `Invoke()`).

- Tento vzor umožňuje použití systémového časovače v okenních aplikacích. Ztratíme tím však jeho milisekundovou přesnost, protože metody přes `Invoke()` se volají bez zaručení rychlosti podle toho, jak rychle první vlákno zvládá obsluhovat své zprávy.

Metoda `Invoke()` je definována v rozhraní `ISynchronizeInvoke`, které implementují především okenní ovládací prvky (všechny třídy odvozené od třídy `Control`). Tato metoda se chová synchronně, tj. volající vlákno určí, co se má zavolat v prvním vlákně, a pak čeká, až toto volání skončí a převezme od něj návratovou hodnotu. Alternativně je však možno použít i asynchronní volání, kdy volající vlákno nečeká na skončení volání v prvním vlákně. Asynchronní model volání je natolik důležitá funkcionalita, že si zaslouží větší pozornost; bude podrobněji popsán v sekci 4.5 na straně 41.

Úloha 15.

Tato úloha je stejná jako předešlá, tentokrát ji však implementujete pomocí systémového časovače a vzoru asynchronní aktualizace GUI.

Vytvořte program zobrazující aktuální čas. Program bude čas (hodiny:minuty:sekundy) vypisovat do okna pomocí ovládacího prvku Label, kterému nastavíte větší font a přizpůsobíte mu i velikost okna. V okně nebude žádné tlačítko, program půjde zavřít křížkem nahoře vpravo.

K zobrazování hodin využijte systémový časovač `System.Threading.Timer`, v obsluze časovače si aktuální čas zjistíte a naformátujete pomocí `DateTime.Now.ToLongTimeString()`. Naformátovaný text předáte do metody třídy okna, která vypíše čas do okna; bude napsaná dle vzoru asynchronní aktualizace GUI, aby šla zavolat i z vlákna časovače.

4.5 Asynchronní programový model (APM)

4.5.1 Co je to APM

O existenci asynchronního programového modelu (APM) jsme se krátce zmínili již v předchozím textu v souvislosti s `Invoke()`. APM se však netýká jen `Invoke()`, jde o obecný model volání metod způsobem, kdy volající nečeká na dokončení volání. Jde tedy o tzv. volání „na pozadí", přitom však nepřicházíme ani o možnost získání návratové hodnoty – pouze je třeba si ji vyzvednout později, až bude volání skončeno.

Základ asynchronního volání jsme de facto již popsali u `Invoke()`, tj. jde o předání delegátu určujícího, co se má zavolat, a seznamu hodnot k dosazení za parametry. Složitost APM začíná až v okamžiku, kdy chceme mít větší kontrolu nad asynchronním prováděním, nebo pracovat i se zmíněnými návratovými hodnotami.

APM je jen „model", definuje jak má vypadat rozhraní umožňující synchronní i asynchronní volání. Funguje to takto: Máme libovolnou metodu jménem `Metoda()`, která pracuje (normálně) synchronně. K ní jsou ve stejné třídě další dvě metody pojmenované `BeginMetoda()` a `EndMetoda()`, tedy přidáním `Begin` a `End` jako předpon názvu. `BeginMetoda()` začne provádět danou akci na pozadí. Má přitom stejné parametry jako `Metoda()` plus dva navíc: Jako předposlední parametr uvedeme callback – to je delegát, který má být zavolán po dokončení asynchronní operace, jako poslední pak uvedeme libovolný uživatelský parametr (má stejný význam jako u zakládání vláken). Jako návratovou hodnotu dostaneme objekt typu `IAsyncResult`, který slouží jako identifikátor této asynchronní operace.

Průvodce studiem

Podobného efektu jako asynchronním voláním lze samozřejmě dosáhnout použitím více vláken. APM je vhodné zejména při práci s I/O (vstup/výstup čili diskové soubory, síťová komunikace atp.), kdy provádění požadovaných operací může i výrazně zpomalovat běh programu. V takových situacích, zejména pak v serverových aplikacích, lze pomocí APM zpracovávat velké množství úloh současně bez potřeby vláken. Při použití vláken by byl systém zatížen poměrně velkou režií s tím spojenou, zatímco asynchronní verze I/O operací používají asynchronní schopnosti operačního systému Windows, který tyto operace umí provádět asynchronně bez použití vláken.

Naopak u malého počtu současně prováděných operací může být použití vláken vhodnější, protože jejich režie v tom případě systém příliš nezatíží a pro programátora to je často srozumitelnější a přehlednější způsob programování

4.5.2 Použití APM

Máme-li tedy nastartovanou asynchronní operaci, tato je prováděna na pozadí a vlákno, které ji spustilo, může dělat libovolnou jinou činnost. Po skončení operace jsme však povinni ještě zavolat `EndMetoda()`, kde jako parametr dáme `IAsyncResult` naší operace a výměnou dostaneme návratovou hodnotu z volání Metoda.

Průvodce studiem

Všimněte si, že asynchronní provádění kódu nás jinak neomezuje v právu na získání návratové hodnoty z operace. Vrací-li metoda nějakou návratovou hodnotu, pak i její asynchronní verze vrací hodnotu stejného typu.

V praxi (zejména mimo .NET) se však můžeme setkat s různými primitivnějšími variantami asynchronního modelu programování, kde se s návratovými hodnotami nepočítá a to pak samozřejmě komplikuje programátorům život. Smyslem APM by však mělo být programy zlepšit, ne zkomplikovat.

Nyní máme tři možnosti, jak se postavit k dokončení operace:

1. Pasivní ukončení (bez čekání). O ukončení se nestaráme. Až operace skončí, sama zavolá callback (námi dodaný delegát, který jsme uvedli jako parametr funkce `BeginMetoda()`). V metodě tohoto delegátu pak zavoláme `EndMetoda()`, čímž operaci řádně ukončíme a získáme její návratovou hodnotu. Callback bude zavolán na vlákně z fondu vláken.
2. Polling (aktivní čekání). Pravidelně testujeme hodnotu příznaku `IAsyncResult.IsCompleted`. Jeho hodnota bude `true`, jakmile bude asynchronní operace dokončena. Tento způsob čekání je nejméně vhodný, neboť je pomalý a zatěžuje procesor neustálým zbytečným prováděním těchto testů. Jakmile `IsCompleted` vrací `true`, voláme `EndMetoda(IAsyncResult)`.
3. Aktivní ukončení (pasivní čekání). Sami zavoláme `EndMetoda(IAsyncResult)` nebo `IAsyncResult.AsyncWaitHandle.WaitOne()`. Toto volání blokuje volající vlákno až do dokončení asynchronní operace.

Důležité upozornění: Ve vašich programech můžete používat libovolný z těchto tří postupů (i když prostřední z nich je méně doporučen), ale pro konkrétní operaci vždy právě jeden z nich.

Je totiž třeba zaručit, že pro každou asynchronní operaci bude právě jednou zavolána metoda **`EndMetoda()`**.

> **Průvodce studiem**
>
> *Nenechte se zmást podobností pojmů „aktivní čekání" a „aktivní ukončení". V prvním případě jde o nejhorší (nejméně efektivní) možný způsob synchronizace a kdykoliv se s ním setkáte, snažte se mu vyhnout. Ve druhém případě však jde o přesný opak – pasivní čekání, kdy volající vlákno během čekání spí a nevyužívá tak procesor vůbec. Termín „aktivní ukončení" je zde použit z hlediska APM, neboť naše hlavní vlákno samo aktivně ukončí asynchronní operaci voláním* **`EndMetoda()`** *a nečeká, až se to stane „samo".*

Úloha 16. Asynchronní kopírování souboru

Typickým příkladem pro APM je čtení či zápis souboru, my vyzkoušíme obojí současně, protože budeme kopírovat soubor. Vaším úkolem je napsat program, který okopíruje daný soubor z jednoho místa na druhé. Přitom jej bude kopírovat po blocích o velikosti 1% délky souboru a během asynchronního kopírování bude na obrazovku vypisovat informace o průběhu kopírování a nějaké další znaky jako signalizaci, že vlákno programu je volné pro vlastní aktivity. (Tyto signalizační výpisy nám budou sloužit jen jako demonstrace, že vlákno je skutečně k dispozici.)

Použijte následující kostru programu, která definuje pomocnou třídu **`Params`** zabalující několik hodnot, které se vám budou hodit v callbacku. Statická metoda **`AsyncCopy()`** provádí vlastní kopírování a mezitím vypisuje informace o jeho průběhu na obrazovku. Soubor je kopírován po 100 blocích, abychom mohli dobře zobrazovat průběh práce v procentech. Zbývá doplnit jen samotný kód pro čtení a zápis souboru.

```
class Parametry {
  public readonly FileStream čtení, zápis;
  public readonly byte[] buffer;
  public int část = 0;
  public Parametry(FileStream čtení, FileStream zápis, byte[]buffer) {
    this.čtení = čtení;
    this.zápis = zápis;
    this.buffer = buffer;
  }
}

static void AsyncCopy(string zdroj, string cíl) {
  using(FileStream čtení = new FileStream(zdroj, FileMode.Open)) {
    byte[] buffer = new byte[(čtení.Length + 99) / 100];
    using(FileStream zápis = new FileStream(cíl,FileMode.CreateNew)) {
      Parametry par = new Parametry(čtení, zápis, buffer);
      ...>>> zde zahájíte asynchronní čtení <<<...
      while(par.část < 100) {
        for(int i = 0; i < 10 && par.část < 100; i++) {
          Console.Write(par.část + "%");
          for(int j = 0; j < 10 && par.část < 100; j++) {
            Console.Write(i == j ? " *" : " .");
          }
          Thread.Sleep(100);
          Console.Write("\r");
        }
      }
      Console.WriteLine();
```

```
        }
    }
}
```

4.5.3 Více o asynchronní práci se soubory

Knihovna .NETu definuje pro proudy základní třídu `Stream`, kde jsou předepsány všechny synchronní i asynchronní operace a je možno je libovolně volat, bez ohledu na to, zda konkrétní implementace (tedy například `FileStream`) či operační systém toto podporuje. Třída `Stream` totiž implementuje synchronní operace tak, že volají své asynchronní protějšky, a opačně.

Každý soubor (přesněji souborový proud) podporuje prostřednictvím třídy `FileStream` synchronní i asynchronní čtení a zápis, operační systém Windows však v jednom okamžiku umožňuje používat jen jedno z toho. Míchání synchronních a asynchronních operací nad jedním souborem je tedy v .NETu možné, není to ale optimální a doporučuje se soubor při změně formy přístupu zavřít a znovu otevřít. Při asynchronním přístupu k souboru je doporučeno použít šestiparametrovou variantu konstruktoru `FileStream`, kde poslední parametr typu `bool` přepíná proud na úrovni operačního systému do asynchronního režimu a tím asynchronní operace zrychlí.

Další zajímavostí může být, že asynchronní operace nejsou použity u požadavků o velikosti do 64KB, čili každé jedno konkrétní volání `BeginRead()`/`BeginWrite()` s délku bufferu do 64KB systém automaticky provede synchronně, protože je to tak obvykle rychlejší.

4.5.4 Asynchronní delegáty

APM lze jednoduše implementovat pomocí delegátů, neboť každý delegát APM již podporuje. Definujeme-li tedy delegát ukazující na naši metodu, můžeme ji pak skrze delegát nechat vykonávat asynchronně. Z hlediska použití jde o klasické APM, oproti výše diskutované práci se soubory je zde však jeden praktický rozdíl: Soubory mají APM implementováno tak, že asynchronní operace jsou prováděny efektivně a bez použití dalších vláken. Naopak „automaticky" vytvořené APM pomocí delegátů používá vlákna z fondu vláken.

Použití asynchronních delegátů je velmi snadné: Místo metody Invoke() jednoduše zavoláme na delegátu `BeginInvoke()`, pro ukončení operace pak `EndInvoke()`.

Na závěr ještě znovu shrňme, které metody asynchronní práce je doporučeno používat: Má-li třída implementováno APM bez vláken (jako např. `FileStream`), dáme mu vždy přednost. V ostatních případech dáme přednost přímému použití funkcionality fondu vláken před voláním přes asynchronní delegáty.

4.6 BackgroundWorker (.NET 2.0)

4.6.1 Popis

Poslední formou asynchronní práce v rámci jednoho procesu, kterou si představíme, je třída `System.ComponentModel.BackgroundWorker`. Tato třída zabaluje vlákno a přidává k němu nové rozhraní umožňující snadné použití v GUI aplikacích. Je k dispozici od .NET Frameworku 2.0.

Jak víme, jen jedno vlákno může aktualizovat GUI prvky (součásti okna), ostatní vlákna ale k oknu mohou přistupovat skrze metodu `Invoke()` pomocí vzoru představeného v kapitole 4.4 na straně 40. Třída `BackgroundWorker` funguje jako vlákno, čili především určíme, jakou metodu chceme vykonat/zavolat tímto vláknem. Zároveň zde však máme k dispozici nástroj pro oznamování stavu operace (doslova progresu, anglicky progress) a to bez ohledu na to, že GUI

se z jiného vlákna volat nedá. Přechod mezi hlavním vláknem a vláknem na pozadí totiž transparentně zajišťuje právě `BackgroundWorker`.

4.6.2 Použití

`BackgroundWorker` je komponenta, tj. je to prvek přímo použitelný ve editoru formulářů (form designeru) Visual Studia. Stejně tak lze založit objekt programově, jednoduše použitím konstruktoru bez parametrů v GUI vlákně. U objektu workeru je po vytvoření potřeba nastavit událost `DoWork` na spouštěcí metodu práce na pozadí. Tuto metodu poté spustíme zavoláním `RunWorkerAsync()` z GUI vlákna a ona vždy poběží ve vlákně workeru. Při spuštění lze také předat volitelný uživatelský parametr. Před spuštěním workeru ale ještě nastavíme událost `ProgressChanged` na metodu GUI, která bude zajišťovat aktualizaci informací o průběhu práce. Tato bude pro změnu volána z workeru, ale poběží v GUI vlákně. Oznamování progresu je třeba ještě povolit pomocí property `WorkerReportsProgress`. (Čili oznamování progresu není povinné.)

Obsluha události `DoWork` má parametr typu `DoWorkEventArgs`, což je třída obsahující v property Argument vstupní parametr předaný při volání `DoWork` a také místo pro uložení výsledku operace v property `Result`.

Bude-li náš worker podporovat také předčasné ukončení práce (neboli zrušení či storno, anglicky cancellation), nastavíme ještě příznak (property) `WorkerSupportsCancellation` na `true`. GUI vlákno potom může kdykoliv zavolat metodu `CancelAsync()` a tím požádat o zrušení operace. Samotné zrušení však musíme provést sami, je proto třeba pravidelně testovat příznak `CancellationPending` (kódem v `DoWork`). Po volání `CancelAsync()` je tento nastaven na `true`.

Po dokončení operace na pozadí skončí metoda `DoWork` a vlákno přestane pracovat (podobně jako u klasického vlákna). Toto lze sledovat pomocí události `RunWorkerCompleted`, která je vyvolána v GUI vlákně. Parametr této události je typu `RunWorkerCompletedEventArgs`, což je třída obsahující tyto užitečné údaje:

- Příznak `Cancelled` je `true`, když došlo k přerušení operace.
  Zde pozor na jednu drobnost: Dojde-li k volání `CancelAsync` těsně před řádným ukončením operace na pozadí, kdy `DoWork` již nepoužije property `CancellationPending` ke zjištění, zda je žádáno zrušení, příznak `Cancelled` je `false`. Čili volání `CancelAsync()` nezaručuje, že tento příznak bude určitě `true`.
- Příznak `Error` je `true`, když operace nebyla dokončena z důvodu neošetřené výjimky.
- Property `Result` nese výsledek operace. Je pochopitelně platný, jen když `Cancelled` i `Error` jsou `false`. (Výsledek musíme nastavit v `DoWork` do property `DoWorkEventArgs.Result`.)
- Property `UserState` nese další uživatelský parametr (nepovinná hodnota, předaná workeru na začátku při volání `DoWork()`).

Poslední zajímavá součást `BackgroundWorkeru` je příznak `IsBusy`, který je `true` po dobu, co worker pracuje.

4.7 Roury

V závěru této kapitoly se krátce zastavíme u rour, což je komunikační prostředek vhodný především k předávání dat tam, kde není možno použít sdílenou paměť. Používání rour nemá moc smysl v rámci jednoho procesu (protože jeho vlákna sdílejí paměť), rovněž mezi aplikačními doménami v jednom procesu je možno komunikovat jednodušeji, než přes roury.

Průvodce studiem

Systém rour je implementován přímo v systému Windows, lze jej tedy používat i ke komunikaci mezi řízeným a neřízeným kódem. Roury lze také použít k síťové komunikaci (mezi počítači v počítačové síti, na všech ale musí být systém Windows). V .NETu přibyla možnost používat tyto systémové roury od verze frameworku 3.5.

Roura je jednosměrný či obou směrný komunikační kanál, kterým „tečou data". Roura má smysl či přínos tehdy, když je každý její konec v jiném procesu. Podobně jako u některých synchronizačních primitiv, roury mohou být pojmenované či nepojmenované. Zde však i nepojmenovaná roura může propojovat různé procesy, neboť roura může fungovat jako standardní vstup či výstup a lze ji procesu „vnutit" přímo při spuštění formou přesměrování vstupu/výstupu vytvářeného procesu do roury.

Průvodce studiem

Právě popsanou funkcionalitu používá operátor `|` *příkazové řádky (svislá čára), který můžeme použít ve formě* `program1|program2`*, čímž specifikujeme, že výstup programu1 má být nasměrován na vstup programu2. Toto funguje úplně stejně ve Windows i v Uniových systémech.*

Roury se v .NETu používají podobně jako souborové proudy, odlišné je především jejich vytváření. Pojmenované roury je možno používat i k propojení procesů na vzdálených počítačích, což žádný jiný prostředek popsaný v tomto textu neumožňuje. Podrobný popis programování s rourami je však nad rámec tohoto učebního textu.

Shrnutí

V této kapitole jsme se věnovali tématům asynchronního programování v .NETu. Kapitola volně navazuje na předchozí, k synchronizačním primitivům jsme tentokrát přidali složitější prvky poskytované knihovnou BCL pro snazší programování typických asynchronních úloh.

Fond vláken je statická třída zjednodušující vytváření vláken a programy hlavně zrychluje, protože objekty vláken i samotná vlákna recykluje namísto toho, aby je opakovaně vytvářel a rušil. Fond je také používán mnoha třídami z BCL (což není vždy jasně vidět, ale je to vždy zdokumentováno v [MSDN]), které provádějí různé servisní a/nebo systémové operace na pozadí. Dále jsme probrali tři verze časovačů, které .NET nabízí – každý je vhodný pro jiné situace. Jeden z časovačů je určený speciálně pro GUI aplikace a v BCL je ještě několik dalších pomůcek pro asynchronní programování s GUI jako třída `BackgroundWorker` implementující vlákno na pozadí či metoda Invoke() pro asynchronní aktualizaci GUI.

Příbuzným tématem k Invoke() je obecný asynchronní programový model (APM), který umožňuje provádět na pozadí či jinými vlákny jakoukoliv operaci. V některých třídách najdeme specializované implementace tohoto modelu, např. třída souborového proudu `FileStream` umožňuje provádět asynchronní čtení i zápis souborů bez použití vláken, což tyto operace zrychluje a umožňuje bez zátěže systému zpracovávat i velká množství požadavků. V závěru kapitoly jsme se ještě zastavili u rour, které jsou pro změnu jistou formou zobecnění proudů.

Pojmy k zapamatování

- Fond vláken
- Asynchronní čekání
- Časovač
- Asynchronní aktualizace GUI
- Asynchronní programový model (APM)
- Pracovník na pozadí (BackgroundWorker)
- Roura

Kontrolní otázky

1. *Jmenujte několik přínosů fondu vláken. Proč je jen jeden takový fond v každém procesu?*
2. *Fond vláken lze využít také pro asynchronní čekání. Jakou má toto řešení výhodu?*
3. *Jmenujte, jaké typy časovačů nabízí .NET. V jakých situacích dáte kterému z nich přednost?*
4. *Vzor asynchronní aktualizace GUI se týká používání vláken v GUI aplikacích. Proč toto potřebuje zvláštní pozornost?*
5. *Popište obecný asynchronní programový model v .NETu.*
6. *Proč je asynchronní práce se soubory s pomocí speciálních metod efektivnější, než když použijeme vlákna a napíšeme si asynchronní operace pomocí nich?*
7. *Vysvětlete přínos asynchronních delegátů a princip jejich fungování.*
8. *Jmenujte výhody background workeru (pracovníka na pozadí) oproti běžnému vláknu.*
9. *K čemu slouží roury?*

5 Paměť a zdroje

Studijní cíle: V této kapitole se budeme věnovat správě paměti v .NETu. Je všeobecně známo, že správa paměti je zde automatická, to však neznamená, že vše funguje tak úplně „samo“ a vždy bezproblémově…

Klíčová slova: garbage collector, řízená halda, generace haldy, sémantika destrukce objektů

Potřebný čas: 100 minut

5.1 Pojem správy paměti

Správa operační paměti je velmi důležitým úkolem každého operačního systému. Jak už víme (či lze nastudovat z [Kep07]), správa paměti zahrnuje velké množství systémových algoritmů, které jsou před programátory aplikačního softwaru skryty. Systém Windows nabízí také poměrně hodně funkcí týkajících se správy paměti ve svém API, naprostá většina programovacích jazyků (i ty starší jako např. jazyk C) však toto složité místo zakrývá a zjednodušuje svou vlastní správou paměti. Kromě Assembleru, který sám nedělá vůbec nic, prakticky každý programovací jazyk nabízí nějakou svou správu paměti a programátoři při práci používají téměř výhradně jen prostředky svého jazyka a přímo na operační systém se neobracejí.

5.2 Automatická správa paměti v .NETu

Jak víme, základní nutnost co do správy paměti, je být schopen přidělovat, evidovat a uvolňovat paměťové bloky. V případě .NETu hovoříme o tzv. „automatické správě paměti“, kde se uživatelské procesy starají jen o alokaci (řeknou systému, kolik paměti potřebují), zatímco evidenci a uvolňování řeší sám systém. Tento způsob správy paměti je velmi odlišný od toho, jaké činnosti s pamětí provádí systém Windows, v důsledku čehož se tato kapitola operačního systému vůbec netýká.

Automatická správa paměti v .NETu má dva hlavní přínosy:

- Snižuje chybovost (čili zvyšuje spolehlivost) kódu
- Odstraňuje fragmentaci paměti

Průvodce studiem

Často omílané zjednodušení kódu je jen vedlejším efektem automatické správy, zejména pro začátečníky je to však k nezaplacení. Je však chybou dávat přednost C# před C++ jen proto, že člověk nechápe správu paměti a chce ji nechat na automatice systému. Taková představa je lichá a vede jen k dalším chybám.

Paměť se v .NETu dělí v zásadě na tři části:

- Zásobník – zde jsou lokální hodnotové proměnné
- Malá řízená halda – zde jsou malé řízené objekty, do velikosti cca 80KB

- Velká řízená halda – zde jsou velké řízené objekty

5.2.1 Organizace malé řízené haldy

Malá halda je místo, kde se obvykle nachází většina objektů. Dělí se na tři generace, které číslujeme 0–1–2. Nově alokované objekty jdou vždy do generace 0, jejíž velikost je obvykle do 16MB.

Při zaplnění paměti vyhrazené pro kteroukoliv generaci se spouští garbage collector (doslova „sběrač smetí“, dále jen kolektor) a ten ji sloučí (neboli provede úklid, sloučení = kolekce, anglicky collect):

1. Jsou nalezeny dožité objekty a jsou zrušeny.
2. Poslední generace je defragmentována, čili živé objekty jsou přeskládány na začátek její paměti. U ostatních generací se živé objekty přesunou o generaci výše.
3. Pokud předchozí bod způsobí zaplnění vyšší generace, i ta je defragmentována.

Jelikož všechny nové objekty jsou umisťovány do generace 0, tato musí být nejčastěji slučována. Přínos generací je právě v tom, že při slučování není třeba uklízet všechny objekty, ale jen ty, které jsou v zaplněné generaci. Šetříme tedy čas a o další generace se nestaráme, dokud se také nezaplní.

Průvodce studiem

Jazyková poznámka: Garbage collector znamená „sběrač smetí“. Slovo collector–sběrač je odvozené od slovesa collect, které znamená sbírat, ale také slučovat. Termín slučování je vhodnější pro české označení operace, kterou garbage collector dělá. Jde stále o jeden a tentýž pojem, jen může být trochu matoucí použití dvou zdánlivě různých českých slov. Při kolekci dochází k vysbírání smetí, což se provede tak, že živé objekty se přesunou na jiné místo v paměti tak, aby byly všechny vedle sebe (na začátku bloku paměti vyhrazeného dané generaci haldy) – tedy jsou sloučeny.

Systém si u každé generace pamatuje pouze její maximální možnou velikost a ukazatel na místo, kam přijde nový objekt – je to vždy přesně za koncem dosud posledního objektu této generace. Každý objekt přitom zabere na haldě přesně tolik bajtů, kolik je jeho velikost (a nic navíc). Ve výsledku je tedy tento způsob paměti na evidenci méně náročný, než systém používaný v prostředích bez automatické správy paměti.

Průvodce studiem

Všimněte si, že použitý alokační algoritmus dodržuje princip časoprostorové lokality: Platí, že společně alokované a společně používané objekty jsou v paměti vždy vedle sebe, a to dokonce i po provedení kolekce.

Generační systém navíc nejčastěji uklízí generaci 0, takže zohledňuje i to, že většina objektů se obvykle ruší brzy po založení. Naopak objekty, které úklid přežijí a dostanou se do další generace, ve které neprobíhá úklid tak často, obvykle žijí ještě déle. (Když už se objekt dožil další generace, lze předpokládat, že asi bude žít dlouho. A to je v souladu s tím, že generace starších objektů se nečistí tak často, protože tam nejspíš stejně žádné dožité objekty nebudou. Tím se činnost kolektoru zrychluje.)

5.2.2 Pinning

Při spolupráci s neřízeným kódem může být důležité mít možnost zakázat některým objektům přesun v paměti během kolekce. Tato možnost tu je a nazývá se pinning (odvozeno od anglického slova „pin“ = připíchnout špendlíkem). Pinning se nezbytný například tam, kde funkce z knihovny BCL volají Windows API a předávají nějaká data odkazem. Kdyby kolektor přesunul objekty během toho, co běží neřízený kód ve Windows API, program by se zhroutil. Pinning vytváří de facto pointer na řízený objekt, kterému říkáme pin (špendlík). Dokud pin nezrušíme, nebude se příslušný objekt v paměti přesouvat.

5.2.3 Hledání dožitých objektů

Důležitou otázkou je, jak vlastně kolektor pozná či najde dožité objekty, když nevedeme jejich seznam, ani nijak zvlášť neevidujeme místa paměti, která jsou obsazená či volná. Základem této operace jsou tzv. aplikační kořeny (application roots) – místa, ke kterým se lze přímo dostat z daného místa aplikace, kde se spouští GC. Živé objekty jsou pak přesně ty, které jsou kořeny nebo na ně z kořenů vede přímý či nepřímý odkaz. GC tedy při hledání dožitých objektů prochází objekty skrze strom jejich odkazů a všechny objekty, které nenavštíví, jsou dožité.

Průvodce studiem

Důležitý rozdíl oproti systémům používajícím počítání referencí je tam, kde několik objektů ukazuje na sebe navzájem. Příkladem může být třeba spojový seznam, kde sousední prvky na sebe navzájem ukazují, takže počítadlo referencí nikdy není nulové, i když celou kolekci třeba už nikdo nepoužívá. V .NETu takovou dožitou kolekci zjistí kolektor velmi jednoduše, protože pokud ji nikdo nepoužívá, nevede na ni žádný odkaz z aplikačních kořenů, takže je možno celou kolekci zrušit.

Aplikační kořeny jsou:

- Všechny globální a statické proměnné
- Všechny lokální proměnné na celém zásobníku
- Argumenty předané do volání metod v celém zásobníku volání
- Všechny registry procesoru odkazující na objekty
- Objekty čekající na finalizaci

5.2.4 Správa velké haldy

Velká řízená halda nemá generace, jinak funguje stejně jako malá. Smysl její existence je v tom, že přesouvat velké kusy paměti zabere více času, takže úklid této hlady provádí kolektor méně často.

5.2.5 Explicitní spuštění kolektoru

Kolektor a automatická správa paměti jsou v .NETu zavedeny proto, aby nám zjednodušily naši práci. Nemá proto smysl se nějak starat o ně, když oni se mají starat o nás. V některých velmi specifických případech však může být žádoucí, abychom sami určovali, kdy se má kolektor spouštět.

Ke kolektoru se dostaneme pomocí statické třídy `System.GC`. Ta nabízí několik metod, z nichž nejdůležitější jsou dvě: Voláním `GC.Collect()` spustíme kolektor. Nepovinně lze uvést číslo

generace, u které má kolekce skončit; standardně se uklízejí všechny generace (odpovídá hodnotě parametru rovné 2). Za toto volání je vhodné přidat ještě volání metody `GC.WaitForPendingFinalizers()`, která počká na zpracování finalizerů – ukončovacích metod, které se vyskytují u některých tříd. Finalizery si podrobněji vysvětlíme v následující sekci.

Průvodce studiem

Kolektor takto explicitně spouštějí například počítačové hry, které potřebují kreslit plynulé video (či grafiku) bez cukání. Jelikož běh kolektoru by program na nějakou dobu zastavil a to by mohlo mít špatný dopad na plynulost videa, hra raději pravidelně kolektor sama volá ve vhodných okamžicích.

5.2.6 Tři verze kolektoru

Distribuce .NET Frameworku ve skutečnosti obsahuje ne jednu, ale hned tři různé implementace kolektoru. Podívejme se nyní na to, jak se mezi sebou liší a jak systém vybírá, kterou z nich na konkrétním počítači použije.

1. Workstation – jednoprocesorová verze
   Tato verze se používá na obyčejných jednoprocesorových počítačích. Kolektor při úklidu zastaví (suspend) všechna řízená vlákna, aby nemohla pracovat s objekty, které právě uklízí. Výhodou je malá režie, protože tento kolektor nepotřebuje řešit synchronizaci. Nevýhodou je, že u realtimových aplikací dochází k viditelnému „cukání“ v okamžicích, kdy kolektor pracuje.
2. Workstation – víceprocesorová verze
   Na víceprocesorových (či vícejádrových) počítačích je generace 0 rozdělena na více částí, tzv. arén, a při alokaci paměti pak každé vlákno alokuje objekty v jiné aréně. Smyslem a výhodou tohoto řešení je, že je možno provádět více alokací současně různými vlákny bez potřeby zamykat haldu. Jelikož na víceprocesorovém počítači mohou teoreticky další vlákna během úklidu haldy vykonávat jiné výpočty, algoritmus úklidu je optimalizován tak, aby ostatní vlákna byla zastavována (suspend) na co nejkratší nutnou dobu.
3. Serverová verze
   Tato verze se používá na serverech. Rovněž předpokládá víceprocesorový počítač a rozděluje haldu na samostatné sekce pro jednotlivé procesory. Úklid pak běží paralelně na všech procesorech současně, každý uklízí svou část haldy. Toto řešení je výhodné pro serverové aplikace.

5.3 Životní cyklus objektu

5.3.1 Obyčejné referenční třídy

Věnujme se nyní instancím referenčních tříd (tedy ne hodnotovým instancím, ani proměnným). Životní cyklus objektů je dán architekturou .NETu, v jednotlivých programovacích jazycích se však může v jistých detailech lišit. Např. v jazyce C++ mohou vznikat nové instance při volání funkcí, bez explicitního vytvoření objektu programátorem. Jazyk C# naštěstí používá mnohem jednodušší model než C++: Objekt vznikne jedině použitím operátoru `new` a je „živý“ tak dlouho, dokud je používán. O odstranění objektu se postará systém automaticky pomocí kolektoru.

Na rozdíl od některých jiných prostředí, např. COM, v .NETu se u objektů nepočítají reference. Na rozdíl od neřízených jazyků, kde se buď sleduje počet referencí, nebo o odstranění objektu požádá sám programátor (např. v C++ operátorem **delete**), v .NETu nevíme, kdy přesně objekt zanikne. Proto taky není zvykem definovat nějaký kód, který by měl být vykonán při rušení objektu (C++ má pro tento účel destruktory).

5.3.2 Třídy používající neřízené zdroje

Specifická situace je u tříd pracujících se systémovými zdroji (či jakýmikoliv neřízenými prostředky), které je třeba explicitně uvolnit. Třídy, které se zdroji pracují, by měly implementovat rozhraní **IDisposable**, které předepisuje jedinou operaci **Dispose()** a tuto je třeba pak explicitně volat při skončení práce s objektem. (Tato metoda uvolní zdroje a samotný objekt existuje dál až do úklidu paměti kolektorem.)

V .NETu lze u třídy definovat i tzv. finalizer, což je metoda, kterou kolektor zavolá při uklízení objektu. (Je zajímavé, že syntaxe definice finalizeru se v jednotlivých programovacích jazycích dosti liší.) Finalizer zhoršuje efektivitu úklidu, takže jej definujeme pouze u těch tříd, které implementují IDisposable. Správný vzor destrukce instancí tříd používajících neřízené zdroje je tento:

1. Definujeme finalizer, v C# se finalizer definuje jako metoda bez parametrů pojmenovaná stejně jako konstruktor s přidáním vlnovky na začátek, tedy např. **~Třída()**. Ve finalizeru korektně ukončíme neřízené zdroje.
2. Implementujeme **IDisposable**. Přidáme proměnnou **IsDisposed** typu **bool**, kterou použijeme ke sledování, zda bylo **Dispose()** již zavoláno.
3. Metodu **Dispose()** implementujeme takto: Ukončíme všechny řízené zdroje, tj. voláme **Dispose()** a přiřadíme null do všech referencí, které jsou **IDisposable**. Potom ukončíme i neřízené zdroje a nastavíme proměnnou **IsDisposed** na true.
4. Do všech veřejných součástí třídy, včetně metody **Dispose()**, přidáme na začátek test proměnné **IsDisposed**. Je-li true, vyvoláme výjimku **System.ObjectDisposedException** (nelze používat zrušený objekt).

Zdrojový kód vypadá takto:

```
class Třída : IDisposable {
  bool disposed;

  public bool IsDisposed {
    get { return disposed; }
  }

  void Úklid(bool disposing) {
    if(!disposed) {
      disposed = true;
      if(disposing) {
        ...úklid řízených zdrojů...
      }
      ...úklid neřízených zdrojů...
    }
  }

  public void Dispose() {
    if(IsDisposed) throw new System.ObjectDisposedException();
    Úklid(true);
    GC.SuppressFinalize(this);
  }

  ~Třída() {
```

```
      Úklid(false);
   }
}
```

Průvodce studiem

Zde uvedený vzor se týká jazyka C# a některých dalších, ne však všech. Například C++/CLI používá jinou sémantiku destrukce, budeme ji diskutovat později.

Řada programátorů přešla k jazyku C# z C++ a zřejmě proto se finalizerům v C# často ne zcela přesně říká destruktory. Jak si ukážeme dále v textu, při srovnání s C++ jsou zde ve skutečnosti jisté odlišnosti.

5.3.3 Vliv finalizeru na život a resurekce (oživování mrtvých) objektů

V předchozí sekci jsme mlčky přešli volání `GC.SuppressFinalize(this)` v metodě `Dispose()`, nyní si jej vysvětlíme.

Kolektor při úklidu dožitých objektů u každého z nich kontroluje, zda implementuje finalizer. Pokud ano, tak je kolekce provedena dvoufázově. Nejprve místo zrušení objektu je tento jen označen k finalizaci a je nechán živý. Toto „označení" znamená, že je objekt připojen do zvláštního seznamu objektů čekajících na finalizaci. Finalizace je pak provedena na pozadí zvláštním finalizačním vláknem až po skončení aktuálního úklidu a objekt zůstává v paměti dokonce až do dalšího úklidu (toto v dané situaci už přímo plyne z logiky věci). Každý finalizovatelný objekt žije (tak trochu zbytečně) o jeden úklid déle, než by bylo třeba jen kvůli tomu, že je finalizovatelný. Proto při volání `Dispose()`, kdy jsou všechny neřízené prostředky uvolněny a následné volání finalizeru tedy již nemá smysl, voláme `GC.SuppressFinalize(this)`, čímž kolektoru oznámíme, že na tomto objektu již finalizaci provádět nemá a má jej hned napoprvé opravdu zrušit.

Tento vzor dvoufázové destrukce ukazuje ještě jednu zvláštní možnost: Pokud bychom to z nějakého důvodu chtěli, ve finalizeru můžeme objekt oživit (resurektovat) – jednoduše jej připojíme zpět k aplikačním kořenům (obvykle se to dělá přiřazením reference do nějaké statické proměnné či kolekce).

5.3.4 Třída agregující objekty používající neřízené zdroje

Pokud naše třída sama s neřízenými zdroji nepracuje a pouze obsahuje objekty jiných tříd, které jsou `IDisposable`, stačí implementovat jednodušší vzor: Naše třída musí být také `IDisposable` a v metodě `Dispose()` zavoláme `Dispose()` u všech agregovaných objektů. `Finalizer` ale nepotřebujeme. Pokud uživatel řádně zavolá naše `Dispose()`, jsou zavolány i `Dispose()` vnitřních objektů a jejich neřízené zdroje jsou správně uvolněny. Kdyby uživatel `Dispose()` na našem objektu zavolat zapomněl, finalizery vnitřních objektů se zavolají tak jako tak při jejich destrukci. Je tedy vidět, že finalizery se v praxi používají méně často, než samotné IDisposable.

5.3.5 Specifika C++/CLI

Jazyk C++/CLI poskytuje bohatší možnosti co do správy paměti. Například umožňuje používání hodnotových instancí referenčních typů, kdy objekty definujeme tak jako v klasickém C++ jako lokální hodnotové proměnné a překladač sám zajistí dodržení referenční sémantiky.

C++/CLI umožňuje používat i neřízené třídy, jejichž instance vytváří na zvláštní neřízené haldě; odpovídá to chování klasického C++, takže C++/CLI má vlastně automatickou i

neautomatickou správu paměti dohromady. Neřízené typy jsou ukončovány stejně jako v C++ operátorem **delete** nebo **delete[]**. U řízených typů se sémantika C++/CLI liší jak od klasického C++, tak od C#.

Deklarujeme-li klasický „vlnovkový" destruktor v řízené třídě, tento je nativně přeložen do metody **Dispose()**, přitom je třída automaticky označena jako **IDisposable**. Tento destruktor můžeme explicitně volat pomocí klasické konstrukce **~Třída()**. Dealokační scénář by měl odpovídat doporučením CLI, čili je třeba deklarovat také finalizer (což odpovídá výše diskutovanému vlnovkovému destruktoru jazyka C#). Finalizer je deklarován podobně jako destruktor, ale s uvedením vykřičníku místo vlnovky před názvem typu.

Následuje ukázka doporučeného řešení:

```
ref class FinClass {
  //destruktor - ukončuje řízené zdroje a volá finalizer
  ~FinClass() {
    ...uvolnění řízených zdrojů...
    this->!FinClass();
  }

  //finalizer - ukončuje neřízené zdroje
  !FinClass() {
    ...uvolnění neřízených zdrojů...
  }
};
```

Průvodce studiem

Všimněte si, že v C++/CLI je tento kód mnohem jednodušší, než v C#.

Na konci destruktoru tedy voláme vlastní finalizer (Pozor! Ze syntaktických důvodů je nutno jej volat odkazem přes **this**.), překladač sám doplní kód zajišťující, aby se po zavolání destruktoru finalizer již nevolal (toto ošetření jsme v C# dělali ručně). Na tomto místě je vidět, že C++/CLI je novější než C# a jeho autoři se poučili z chyb v návrhu jazyka C#. Koncept destrukce objektů v C++/CLI je totiž jednoznačně lepší a jako takový by se jistě hodil i do příští verze jazyka C#. Základní koncepční rozdíl lze popsat takto: C# vznikl v době kdy mezi odborníky převládalo přesvědčení, že správný princip zní: „Kolektor místo destruktoru." O několik let novější jazyk C++/CLI již funguje na principu: „Kolektor a destruktor." a praxe ukazuje, že tento model správy paměti je lepší.

S modelem destrukce souvisí i výše zmíněná možnost používat hodnotové proměnné řízených referenčních typů, které můžeme vytvořit jak v rámci metody, tak v rámci jiné třídy. Příklad následuje.

```
ref class A { ... };
ref class B {
 ...
 A a; //lokální instance typu A
};

void Metoda() {
  A a; //lokální instance typu A
  B b; //lokální instance typu B
}
```

C++/CLI u těchto proměnných zajišťuje automatické volání destruktorů v okamžiku ukončení nadřazeného bloku kódu či objektu. I tento prvek jde za hranice C#, C++/CLI je zřejmě dokonce prvním jazykem umožňujícím tento pohodlný způsob vytváření instancí a řízené destrukce objektů referenčních typů. V jazyce C# lze toto pouze částečně nahradit blokovým příkazem using.

Jazyku C++/CLI je věnován také článek [Kep06].

5.4 Další témata

5.4.1 Chování systému při nedostatku paměti

Současné počítače mají dost paměti na to, aby většina programů nemusela nikdy nedostatek paměti potkat a tudíž ani řešit. Přesto se však můžeme podívat, co se v systému při nedostatku paměti děje.

Nedostatek paměti může nastat ve dvou případech: Buď je plná halda, nebo je plný zásobník. V případě nedostatku místa pro vytvoření nového objektu na haldě se aktivuje kolektor a ten získá další paměť úklidem. Pokud ani po úklidu není paměti dost, je vyhozena výjimka `System.OutOfMemoryException`. Tuto výjimku lze běžným způsobem zachytit pomocí `try–catch` bloku a ošetřit. Během hledání příslušného `catch` bloku se mohou spouštět mezilehlé `finally` bloky a ty mohou uvolňovat objekty. Je tedy možné, že v místě zachycení výjimky bude původně chybějící paměť již k dispozici (ačkoliv z toho obvykle nelze nic vytěžit, protože došlo mezitím ke ztrátě jiných objektů). Pravidla .NETu také vyžadují (a to je důležité!), aby se při uvolňování objektů nic nového nealokovalo. Tzn. finalizer a `Dispose()` nesmějí nic alokovat, takže při úklidu paměti nemůže dojít k zacyklení neustálým vyhazováním výjimky `OutOfMemoryException`.

Druhá možnost zaplnění paměti se týká zásobníku. Připomeňme, že u programového zásobníku se při kompilaci klasickým způsobem (tj. stejně jako u nativního kódu) nastavují hodnoty committed a reserved, tedy jeho počáteční a maximální velikost. Každé vlákno má nejprve zásobník o velikosti committed a při jeho zaplnění se tento automaticky zvětšuje až na velikost reserved. Při maximálním zaplnění a dalším nedostatku je vyhozena výjimka `System.StackOverflowException`. Alokace místa na zásobníku se přitom provádí jen na začátku metody – tehdy se vytvoří místo pro všechny lokální proměnné definované v této metodě bez ohledu na to, jaký přesně je jejich rozsah platnosti (ten totiž může být menší, než na celou metodu). Výjimka `StackOverflowException` znamená, že program nemůže dál pokračovat. Tuto výjimku nelze zachytit v `try–catch` bloku, program je totiž okamžitě ukončen pomocí `System.Environment.FailFast()`. Běhové prostředí pouze umožňuje nastavit, aby systém při `StackOverflowException` ukončil jen aplikační doménu, kde chyba vznikla, a zbytek procesu zůstal běžet. Toto je tedy jediná alternativa k okamžitému ukončení celého procesu, navíc není k dispozici na Windows 9x. Více viz [MSDN] či [Duf06].

5.4.2 Paměťová brána

Čistější způsob, jak se vypořádat s možným nedostatkem paměti, nabízí třída `System.Runtime.MemoryFailPoint`. Vytvořením objektu této třídy vzniká tzv. paměťová brána (memory gate), velikost požadované paměti v megabajtech zadáme jako parametr konstruktoru. Objekt brány je úspěšně vytvořen, jen když je v systému dostatek paměti (dle parametru konstruktoru), v opačném případě tato operace skončí výjimkou.

Vytvořením brány systém nezaručuje dostatek paměti dlouhodobě, ale jen v okamžiku jejího vytvoření. Pokud však všechen kód alokující paměť (a to především velké bloky) používá brány, tak díky tomu, že brány jsou `IDisposable`, systém zohlední celkovou kapacitu

potřebnou všemi otevřenými branami. Zavoláním `Dispose()` se brána uzavře a dáme tím signál, že danou paměť již nepotřebujeme rezervovat. Vzor použití brány vypadá takto:

```
using (MemoryFailPoint gate = new MemoryFailPoint(kolik)) { ... }
```

Při volání konstruktoru se tedy zjistí, zda je paměti dostatek. Při nedostatku se postupuje takto:

1. Je provedena kolekce všech částí řízené haldy.
2. Pokud stále není paměti dostatek, systém se pokusí zvětšit swap file ve Windows.
3. Pokud to stále nestačí, je vyhozena výjimka `InsufficientMemoryException`. (Je to tedy jiná výjimka, než `OutOfMemoryException` vyhozená při skutečném nedostatku paměti v okamžiku vytváření objektů.) Program může tuto výjimku zachytit, je to signál, že blok kódu, který měl pracovat v rámci brány, ani nezačal. To je často daleko bezpečnější, než složitější operaci začít a teprve v jejím průběhu zjistit nedostatek paměti – nemusíme totiž řešit úklid a zotavení systému z nekonzistentního stavu.

Poznámka: Paměťové brány samozřejmě mají efekt či přínos pro naše programy jen tehdy, když v systému nejsou jiné procesy (či vlákna stejného procesu) alokující paměť ve velkém množství bez bran. Brána totiž sama žádnou paměť nerezervuje a pokud těsně po jejím otevření jiný proces „seber" všechnu paměť pro sebe, má náš proces jednoduše „smůlu".

Shrnutí

V této kapitole jsme se seznámili se správou paměti a zdrojů a zejména pak nahlédli pod pokličku automatické správy paměti v .NETu. V první části kapitoly jsme se seznámili s tím, jak správa objektů funguje a především jsme se zabývali (garbage) kolektorem, který se stará o úklid dožitých objektů z paměti a slučování objektů živých. Ve druhé části kapitoly jsme se naučili, jak správně programovat třídy používající neřízené zdroje (a to přímo, či nepřímo přes agregaci jiných objektů). V závěru jsme ještě nakoukli k jazyku C++/CLI, který přestože je syntakticky podobný, používá zcela jinou sémantiku destrukce objektů a souvisejících věcí.

Pojmy k zapamatování

- Automatická správa paměti
- Malá a velká řízená halda
- Pinning
- (Garbage) kolektor
- Životní cyklus objektu
- Finalizer
- Resurekce objektu
- C++/CLI
- Paměťová brána

Kontrolní otázky

1. *Vysvětlete, proč se správa paměti v .NETu nazývá „automatická".*
2. *Které dvě vlastnosti jsou uváděny jako hlavní přínosy automatické správy paměti?*
3. *Co je to řízená halda a na které čtyři části se v .NETu dělí? Charakterizujte je.*
4. *Popište algoritmus práce (garbage) kolektoru v .NETu.*
5. *Co je to pinning?*
6. *Platforma .NET má tři různé verze kolektoru. Vysvětlete rozdíly mezi nimi a kdy se který použije.*
7. *Popište vliv finalizeru na životní cyklus objektu.*

8. *Popište rozdíly mezi životním cyklem objektů v jazycích C# a C++/CLI.*
9. *Jak se systém chová při nedostatku paměti na zásobníku a haldě? Proč se tyto dva případy liší?*
10. *Co je to paměťová brána? Popište situaci, kde z principu nelze použít a vysvětlete proč.*

A Windows – kapitola nula

Studijní cíle: Tato příloha se zabývá dvěma základními vlastnostmi systému Windows, které nepřímo ovlivňují i chování aplikací běžících v .NETu. Představíme si systém chybových kódů, systém identifikátorů objektů a způsob práce se znaky a textem.

Klíčová slova: HRESULT, HANDLE, MBC, unicode, UTF-16

Potřebný čas: 45 minut

A.1 Windows API

Operační systémy z rodiny Windows NT (včetně Windows 2000, XP, Vista a všech Windows Serverů) fungují na bázi univerzálního systémového jádra NT, ke kterému přímo aplikace (uživatelské programy) nepřistupují. Místo toho systém obsahuje několik tzv. subsystémů a každý z nich nabízí aplikacím své rozhraní. Důvodem této systémové architektury je snaha umožnit kompatibilitu s různými staršími systémy, které se používaly před vznikem Windows NT na začátku 90.let 20.století. Většina součástí rozhraní NT jádra není dodnes veřejně zdokumentována.

Dnes jednoznačně nejpoužívanějším je subsystém Win32 a jeho aplikační programové rozhraní nazývané Win32 API. V posledních letech s příchodem 64bitových verzí Windows se toto rozhraní přeneslo také do 64bitové podoby v novém subsystému Win64. Microsoft na úrovni zdrojových kódů tyto dvě API sjednotil a začal je nazývat jednotným „Windows API". Programujeme-li tedy pro Windows v některém nativním jazyku (C, C++, Delphi apod.), používáme funkce Windows API bez ohledu na to, zda jde o programy 32bitové nebo 64bitové.

Průvodce studiem

Snaha Microsoftu o sjednocení API subsystémů Win32 a Win64 nedopadla úplně úspěšně. Ačkoliv z hlediska zdrojových kódů je současné Windows API skutečně jednotné, 64bitová verze používá paměťový model LLP64 a kvůli tomu jsou 64bitové programy vlastně pořád napůl 32bitové. 64bitové jsou v něm totiž jen pointery a málo používané `long long` *proměnné, zatímco* `int` *i* `long` *zůstávají 32bitové. Většina proměnných ve výsledných 64bitových programech tedy je 32bitových. V systému Linux, který také má 64bitovou verzi, se naproti tomu používá paměťový model LP64, kde i* `long` *má 64 bitů. Stejné či hodně podobné jako v Linuxu je to i v Unixu. (Tyto paměťové modely se samozřejmě týkají jen nativního programování a nemají vliv na fungování programů v rámci .NETu.)*

Platforma .NET není plnohodnotným subsystémem NT, funguje totiž jako nadstavba na subsystémem Win32 či Win64. V okamžiku spuštění nějakého „exe" souboru, který má běžet v .NETu si jeho inicializační kód (uložený v tomto „exe" souboru) sám spustí běhové prostředí CLR, ve kterém se pak kód .NETu vykonává. Programátor v .NETu se s Windows API setkává jen výjimečně, obvykle v místech, kde návrh knihovny .NETu není úplně dotažen. V dalším textu této kapitoly si stručně představíme dva koncepty, které jsou pro Windows podstatné a jejich znalosti by se programátorovi .NETu mohla hodit.

A.2 Výsledkové kódy

Windows API je jazykově neutrální rozhraní nepoužívající třídy. Z těchto důvodů je snadno volatelné prakticky odkudkoliv, ale na druhou stranu není moc přehledné, protože je to doslova „hromada" globálních funkcí, kde každá má navíc obvykle poměrně hodně parametrů (například deset).

Základním společným rysem většiny funkcí Windows API je, že o výsledku informují prostřednictvím návratové hodnoty typu HRESULT. Tento typ je 32bitové znaménkové číslo, kde záporné hodnoty jsou chybové kódy a nula je bezchybný výsledek. Některé funkce mohou vracet také různé kladné hodnoty, což jsou nějaké zvláštní nechybové stavy (například upozornění (anglicky warning)). Některé funkce API mohou místo HRESULT vracet přímo výsledek operace a v dokumentaci je pak uvedeno, jakou hodnotu vracejí při chybě (paradoxně to může být i nula, která je v případě HRESULT označením bezchybného výsledku).

Platforma .NET používá pro oznamování chyb systém strukturovaných výjimek, který má jak víme mnoho výhod. Zejména je bezpečnější, protože každou chybu musí někdo zachytit a nějak zpracovat, dále kód je přehlednější, protože chyby nemusíme neustále testovat neustálým opakováním „if" příkazů a kód je díky tomu také samozřejmě mnohem kratší a přehlednější. Některé třídy .NETu však používají funkcionalitu operačního systému, nejčastěji proto, že jinak by daná operaci jednoduše nešla vykonat. Například se to týká práce se soubory, protože bez funkcí Windows API by .NET jednoduše neměl jak se soubory pracovat. V těchto místech, kde knihovna .NETu volá funkce Windows API, se teoreticky může stávat, že API vrací HRESULT hodnotu, které .NET nerozumí a místo zpracování ji předá přímo do aplikace. (A nemusí to být zrovna u práce se soubory, to byl jen příklad volání API z .NETu.) V našich programech se nám pak objeví nespecifikovaná výjimka, kde v jejím textu vidíme nějaký chybový kód vypsaný obvykle v šestnáctkové číslo. Jelikož chybové kódy jsou záporná čísla, první cifra je vždy v rozmezí 8–F hexa, např. tedy 0x8000000. Vysvětlení tohoto chybového kódu pak je třeba hledat v [MSDN], kde je Windows API zdokumentováno.

Průvodce studiem

Součástí Visual Studia je aplikace „Error lookup", která po zadání čísla chyby Windows API vypíše její popis. Najdete ji jako errlook.exe, obvykle v adresáři Common7.

Windows API má dvě užitečné funkce: `GetLastError()` vrací poslední chybový kód aktuálního vlákna, pomocí funkce `FormatMessage()` pak můžeme získat i textový popis této nebo kterékoliv jiné systémové chyby.

Podrobnější informace o HRESULT lze najít v [MSDN] nebo také v [Wiki].

A.3 Identifikátory objektů (hendly)

Ačkoliv Windows API nepoužívá přímo třídy a objekty, používá systém datových struktur, který je nápadně připomíná (ale nepodporuje dědičnost, typovou kontrolu atp.). Systémové funkce mají často jako první parametr tzv. „hendl" (anglicky handle, česky rukojeť/držadlo, ale obvykle se říká slangově hendl). Hendl je tedy identifikátorem objektu, se kterým má funkce pracovat. Jazykově neutrální rozhraní nemá nástroj, jak zjistit či dokonce vynutit datové typy, takže typová kontrola se neprovádí. Hendl si tedy můžeme představit spíše jako netypový ukazatel (pointer).

Základním typem pro hendly je `HANDLE`, v jazyku c je definován jako `void*`. Další typy objektů pak mají definovány další typy s názvem charakterizujícím, o co jde, plus písmeno H na

začátku. Například tedy existuje `HICON`, `HBITMAP`, `HFILE`, `HMENU` atd. Všechny tyto typy jsou ale opět jen další názvy pro netypový ukazatel `void*`. (Smysl to má tehdy, pokud překladač programovacího jazyka rozlišuje různě pojmenované netypové pointery jako odlišné typy.)

Programátor neví, co je uvnitř těchto datových struktur, protože jednak nezná jejich formát, a jejich data také mohou být v paměti jádra, kam se nelze přímo dostat. I v .NETu se tyto hendly zákonitě projeví, protože například každý soubor, se kterým pracujeme, si ve svém objektu musí pamatovat hendl souboru v operačním systému. Je pak na autorech té které třídy v .NETu, jak moc odkryjí či zakryjí fakt, že někde v pozadí má objekt jejich třídy alespoň jeden systémový hendl. Všechny třídy objektů uchovávající systémové hendly implementují rozhraní `IDisposable`, protože hendl je neřízený systémový prostředek, o který se nestará automatická správa paměti .NETu. Při skončení práce s těmito třídami tedy musíme volat `Dispose()`.

Na závěr ještě dodejme, že v systému Windows se po skončení práce s hendlem musí volat `CloseHandle()`, některé typy hendlů však vyžadují volání speciálních uzavíracích funkcí podle svého typu.

A.4 Znaky a texty

Systém Windows NT byl od začátku navržen tak, aby používal dvojbajtové znaky unicode. Díky tomu je tam dobře podporováno národní prostředí, ať už jde o jakýkoliv národ. Zároveň však Windows NT umí používat i starobylé jednobajtové znaky, což je nutné pro kompatibilitu se starými verzemi Windows (ne–NT) a MS-DOS.

Windows API je navrženo jako znakově neutrální na úrovni zdrojového kódu C/C++. V těchto jazycích tedy lze napsat jedinou verzi programu a překladač ji přeloží do jednobajtové formy, či unicode. Podívejme se však na to, jak se znaky a texty nativně pracuje Windows.

Windows API rozlišuje několik variant kódování textu:

- Klasické jednobajtové. Význam znaků 128–255 je dán tzv. „kódovou stránkou“, která je specifická vždy pro skupinu příbuzných národních prostředí. Čeština je zahrnuta do stránky 1250, spolu s dalšími východoevropskými jazyky a angličtinou.
- Vícebajtové (MBCS). Toto je rozšířením předchozího, je vhodné tam, kde mají v národních abecedách velmi mnoho znaků a do 256 by se všechny nevešly. (Pochopitelně, MBCS se v praxi nejvíc používá ve východní Asii.) Každý znak má jiný počet bajtů (anglické vždy jeden, národní více) a tyto jsou kódovány pomocí tzv. escape sekvencí, kde první bajt je značkou rozšířeného znaku a další bajt jej upřesňuje.
- Dvojbajtové unicode. Jde o starší kódování UCS-2 dle staršího standardu unicode, kde každý znak měl právě dva bajty.
- Vícebajtové unicode. Jde o kódování UTF-16 dle aktuálního standardu unicode, kde každý znak má 2 nebo 4 bajty

Jak je patrno s popisu, první dvě a druhé dvě varianty de facto splývají ve dva přístupy: jednobajtový a dvojbajtový. Windows API nabízí poměrně velké množství funkcí pro práci s textem, včetně bohaté funkcionality týkající se změny kódování, zjišťování vlastností znaků a textů (např. ke zjištění délky unicode řetězce nestačí jen vydělit délku v bajtech dvěma). To, jestli i konkrétní uživatelský program podporuje i druhou či čtvrtou variantu, nebo jen první či třetí, závisí na tom, jakým způsobem se tam s textem pracuje. Tj. když například délku textu bere podle délky v bajtech, pak pro 4bajtové unicode znaky nefunguje. Unicode také například umožňuje, aby tentýž znak měl různé binární kódy (tj. různá čísla znamenají stejný znak), takže ani porovnávání na rovnost pomocí rovnosti bajtů nemusí fungovat.

Implementace

Podpora různých kódování znaků a textu je ve Windows API řešena bohatou sadou systémových funkcí, jak již bylo zmíněno výše. To ale není vše. Každá systémová funkce Windows API, která má ve svých parametrech text či znaky, má ve skutečnosti definovány dvě verze: Jedna má na konci názvu A a pracuje s MBCS, druhá má na konci názvu W a pracuje s UTF-16.

Programátoři tuto dvojakost rozhraní Windows API přímo nevidí, protože dle nastavení překladače se jim nabízí jen jedna z těchto funkcí. Např. k vytvoření či otevření souboru se používá funkce `CreateFile()`, která samozřejmě přijímá jako parametr i jméno souboru, pracuje tedy s textem. Windows API definuje funkce `CreateFileA()` a `CreateFileW()`, přičemž `CreateFile()` je ve skutečnosti jen makro nasměrované na jednu z dvou skutečných funkcí.

Průvodce studiem

Překladače C/C++ používají jako výchozí variantu unicode. Chceme-li programovat s klasickým jednobajtovým textem (což je zřejmě obvyklý případ), je třeba v nastavení projektu přepnout kódování překladače na MBCS. Použitím MBCS sice programy omezíme v oblasti podpory národních textů, avšak obvykle si tím výrazně zjednodušíme programátorský život, protože běžný programátor C/C++ unicode není zvyklý používat.

Dodejme, že samotné standardy C/C++ s unicode verzemi počítají, ovšem v operačních systémech mimo Windows se toto prakticky vůbec nepoužívá.

Windows 95, 98 a Me přímo unicode nepodporují, jsou v nich tedy jen funkce zakončené na A a makra na ně směřují stále. Do těchto systémů je však možno doinstalovat nástavbu, která přidá základní podporu unicode k vybraným funkcím a umožňuje tak používat unicode texty i na těchto systémech. Základní podpora přímo v systému se zde omezuje na funkce překódování textu z/do unicode, zmíněná nástavba je pak součástí překladače, potažmo naší aplikace (přeloženého programu).

Shrnutí

Tato přílohová kapitola poskytuje základní informace o Windows API, jejichž znalost (alespoň obrysová) se hodí i při programování v .NETu, který je de facto nástavbou nad Windows. Diskutovány jsou návratové kódy (HRESULT), identifikátory objekty (HANDLE) a systémová podpora pro práci s texty ve formátu unicode.

Pojmy k zapamatování

- Windows API
- Výsledkový kód (HRESULT)
- Identifikátor objektu (HANDLE)
- Kódování textu MBCS a unicode

Kontrolní otázky

1. *Co je to Windows API? V jakých systémech je k dispozici?*

2. *Vysvětlete základní pravidla pro výsledkové kódy ve Windows (tj. které skupiny čísel znamenají co).*
3. *K čemu ve Windows slouží tzv. hendly?*
4. *Které varianty kódování znaků Windows podporuje? Proč musí být podpora práce s textem obsažena přímo v operačním systému? (Nebo nemusí? Vysvětlete.)*

B Thread Local Storage

Studijní cíle: Tato příloha seznámí čtenáře podrobněji s problematikou Thread Local Storage (TLS). Jedná se o přepis článku [Kep05] doplněný o pár upřesňujících faktů.

Klíčová slova: TLS, thread local storage, Windows, Linux

Potřebný čas: 75 minut

Všechna vlákna v rámci jednoho procesu sdílejí stejný adresový prostor. Výjimkou je Thread Local Storage (TLS), lokální datový prostor každého vlákna. TLS je vhodným doplňkem objektově orientovaného programování. Princip, kterým umožňuje navzájem oddělit proměnné (obecně paměť) v jednotlivých vláknech, je možné nahradit i klasickými prostředky OOP, ale s pomocí TLS můžeme psát některé konstrukce paralelních výpočetních programů jednodušeji, stručněji a přehledněji.

B.1 Úvod

Vícevláknové programování není žádnou novinkou. Většina běžných aplikací ve Windows ale i dalších operačních systémech dnes používá více než jedno vlákno, nejčastěji pro zlepšení odezvy uživatelského rozhraní (na povely uživatele). Jelikož adresový prostor je vztažený vždy k jednomu procesu, všechna vlákna v procesu sdílí stejnou paměť. To má za následek, že všechny globální proměnné, včetně statických proměnných tříd a metod, jsou sdílené mezi všemi vlákny. V běžném scénáři, kdy jednotlivá vlákna mají v rámci aplikace odlišné úlohy, je tento fakt výhodou, neboť každé vlákno pracuje s odlišnými daty a sdílenou paměť je možno využít k rychlé mezivláknové komunikaci.

S příchodem dvou– a vícejádrových mikroprocesorů a jejich postupným rozšiřováním na běžné počítače se začal objevovat i jiný druh vícevláknových aplikací – více vláken je nasazeno na urychlení jedné operace, provádějí tedy společně výpočet jedné úlohy, čili pracují se stejnými daty. U vícevláknových aplikací tohoto typu se objevuje potřeba rozlišovat data sdílená mezi vlákny a data vláknu vlastní. Způsobů, jak zajistit, aby vlákno mělo k dispozici kromě sdílené paměti také nějakou vlastní paměť nezávislou na ostatních vláknech, je hned několik. Běžné operační systémy, jako Windows nebo Linux, nabízejí pro tyto účely tzv. Thread Local Storage (TLS), česky „lokální paměť vlákna. TLS je možno využít prostřednictvím funkcí operačního systému; některé překladače C/C++ (a pravděpodobně i jiných jazyků) podporují TLS přímo a umožňují tak používat TLS velmi jednoduchým a přehledným způsobem. Lokální paměť vlákna je možno také simulovat pomocí čistých objektově–orientovaných konstrukcí, což je však nepraktické a vyžaduje složitější kód.

V následujících kapitolách bude představeno několik způsobů, jak lokální paměť vlákna realizovat.

B.2 Kontext – lokální paměť vlákna bez podpory operačního systému

Zdánlivě nejjednodušší způsob, jak dosáhnout toho, aby vlákno mělo vlastní paměť, nepotřebuje žádnou podporu operačního systému. Stačí všechna data, která chceme mít duplikována pro každé vlákno, umístit do samostatné třídy a ve spouštěcí funkci vlákna pak vytvořit vždy jednu novou instanci této třídy. Referenci tohoto objektu pak předáváme jako „kontext“ do každé metody, která lokální data vlákna používá nebo volá jiný kód, který by je používat nebo potřebovat mohl.

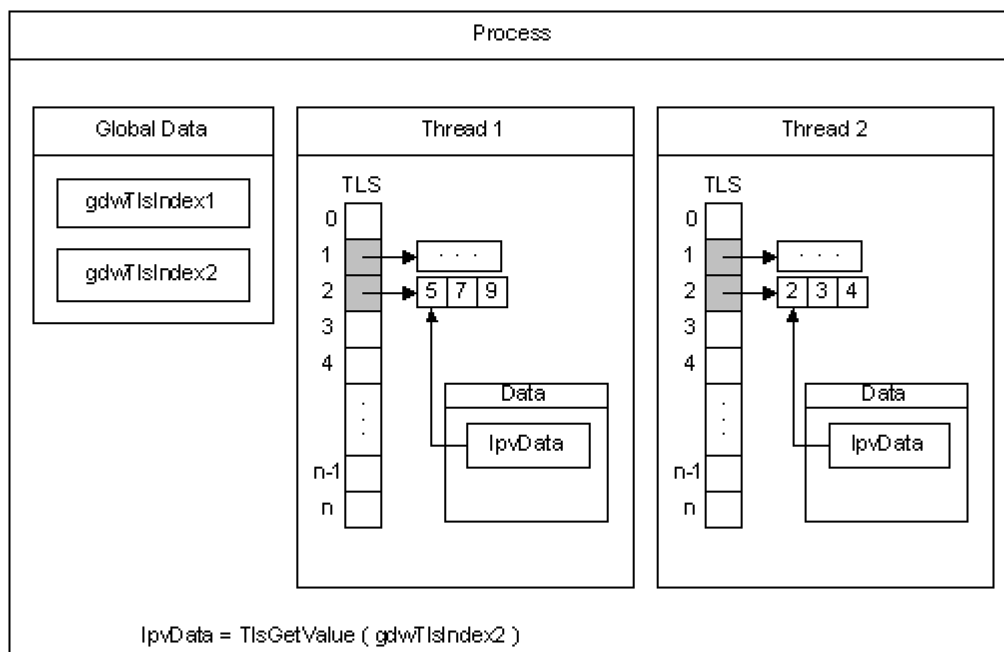
Výhodou tohoto řešení je nezávislost na platformě. Nevýhodou je nutnost předávat kontext jako parametr do každé metody (ve všech třídách celé aplikace), což může být navíc nerealizovatelné při používání cizích knihoven, inverzním programování (vkládání vlastního kódu do kódu cizích knihoven, např. iterátory v jazyce C#), apod. Další nevýhodou je, že je zde porušen princip odpovědnosti tříd, neboť některá data jsou z čistě implementačních důvodů vyčleněna mimo třídy, kam by měla patřit, do třídy kontextu. Je-li program rozsáhlejší, vzniká tím navíc velká nepřehlednost. Tu lze částečně zlepšit použitím kaskádových kontextů, tj. shlukováním souvisejících lokálních proměnných do tříd, jejichž instance teprve umístíme do třídy kontextu. Nevýhodou může být rovněž o jednu úroveň vyšší nepřímost adresování (proměnná není přímo v „mé" třídě, dostanu se k ní až pomocí reference na jiný objekt), což může výrazně zpomalit časově náročné výpočty (platí o to více u kaskádových kontextů).

B.3 TLS s podporou operačního systému

Oproti čistě objektovému řešení popsanému v předcházející kapitole, TLS s podporou operačního systému umožňuje používání lokální paměti vláken bez nutnosti předávat do všech metod navíc tzv. kontext jako další parametr.

B.3.1 Windows API (Windows 95/98/NT/2000/XP/Vista)

Princip je následující: Jedno vlákno (obvykle hlavní/primární vlákno procesu) alokuje tzv. TLS–index pro každou proměnnou, kterou chceme mít lokální pro každé vlákno. TLS–index je číslo, které si musíme uložit do globální proměnné přístupné všem vláknům. Lokální proměnné vlákna tak dostanou číselné indexy, pomocí kterých je můžeme (a musíme) identifikovat. Index proměnné je stejný ve všech vláknech, ale adresa proměnné na tomto indexu je v každém vlákně jiná. Rámcově je tento princip nastíněn na následujícím obrázku .



Obr. 2 TLS ve Windows [MSDN]

Každé vlákno má TLS tabulku, což je právě ona lokální paměť vlákna. Tato tabulka je inicializovaná nulami a její velikost je vždy stejná, závisí jen na verzi operačního systému (Windows 95/NT4.0 má 64 buněk, Windows 98/ME má 80 buněk, Windows 2000/XP má 1088 buněk). Do této paměti máme de facto volný přístup pomocí příslušných funkcí operačního

systému, je však doporučeno ji adresovat pomocí oněch TLS–indexů. Každá buňka TLS tabulky je typu LPVOID, čili je to 4bajtová hodnota. Potřebujeme-li ukládat více než 4 bajty, ukládáme do TLS pouze pointery na skutečná data, která umístíme do běžné paměti. Proměnné o délce do 4 bajtů můžeme do TLS ukládat přímo.

Postup:

1. Hlavní vlákno alokuje TLS–index pro lokální proměnnou vláken.

   ```
   static DWORD index = TlsAlloc();
   ```

2. Každé vlákno při své inicializaci vytvoří lokální proměnnou (zde pro příklad typu MyClass – do TLS ukládáme pointer na skutečný objekt).

   ```
   TlsSetValue(index, new MyClass());
   ```

3. Při práci se svou lokální proměnnou si vlákno nejprve přečte pointer na svůj lokální objekt z TLS. Pokud s ním bude pracovat déle, je vhodné si jej uložit do běžné lokální proměnné metody (tj. na zásobník, mimo TLS).

   ```
   MyClass *objekt = (MyClass*)TlsGetValue(index);
   ```

4. Při ukončování vlákna je třeba uvolnit alokovaný objekt.

   ```
   delete (MyClass*)TlsGetValue(index);
   ```

5. Po skončení všech pracovních vláken pak hlavní vlákno uvolní TLS–index. (Není nutno dělat při ukončování celého procesu.)

   ```
   TlsFree(index);
   ```

B.3.2 POSIX a Linux

V Linuxu můžeme používat TLS pomocí knihovny pthreads (POSIX Threads), viz [Bar05]. Postup práce je stejný jako ve Windows, liší se jen jména systémových funkcí. Výjimkou je možnost definovat destruktory (viz níže).

Postup:

1. Hlavní vlákno alokuje TLS–index pro lokální proměnnou vláken.

   ```
   #include <pthread.h>
   pthread_key_t *key;
   pthread_key_create(key,NULL);
   ```

   Jako druhý parametr je možno předat pointer na destruktor – funkci, která je volána při ukončování vlákna pro všechny nenulové buňky TLS tabulky.

   ```
   void (*destructor)(void*);
   ```

2. Každé vlákno při své inicializaci vytvoří lokální proměnnou (zde pro příklad typu MyClass – do TLS ukládáme pointer na skutečný objekt).

   ```
   pthread_setspecific(index, new MyClass());
   ```

3. Při práci se svou lokální proměnnou si vlákno nejprve přečte pointer na svůj lokální objekt z TLS. Pokud s ním bude pracovat déle, je vhodné si jej uložit do běžné lokální proměnné metody (tj. na zásobník, mimo TLS).

   ```
   MyClass *objekt = (MyClass*)pthread_getspecific(index);
   ```

4. Při ukončování vlákna je třeba uvolnit alokovaný objekt.

   ```
   delete (MyClass*)pthread_getspecific(index);
   ```

5. Po skončení všech pracovních vláken pak hlavní vlákno uvolní TLS–index. (Není nutno dělat při ukončování celého procesu.)

   ```
   pthread_key_delete(index);
   ```

Zde popsané fungování TLS v Linuxu je možno použít na všech systémech podporujících POSIX (viz [Bar05]). Narozdíl od TLS ve Windows umožňuje POSIX používání destruktorů, což je velmi praktické, neboť to zjednodušuje uvolňování dynamicky alokovaných objektů odkazovaných z TLS. Knihovnu pthreads je možno používat, a tím dosáhnout zde popsané funkcionality, i v jiných systémech než jen v Linuxu (včetně Windows, není to však obvyklé).

B.3.3 Platforma .NET

Platforma .NET je prostředím, ve kterém jsou spouštěny aplikace, stojí tedy z hlediska aplikací na úrovni operačního systému. Proto také sama nabízí funkcionalitu TLS a popis je zařazen do této kapitoly k běžným operačním systémům. TLS v .NETu je možno používat ve všech programovacích jazycích, příklady jsou zde uváděny v jazyce C#.

Platforma .NET je plně objektová, místo číselných TLS indexů se zde používají objekty typu **`LocalDataStoreSlot`**, které jsou v terminologii .NETu nazývány sloty. Samotná funkcionalita je však v zásadě stejná jako ve Windows či Linuxu (viz také [Doe03]).

Použití:

1. Hlavní vlákno alokuje TLS slot pro lokální proměnnou vláken.

   ```
   LocalDataStoreSlot slot = Thread.AllocateDataSlot();
   ```

   Alternativně je možno TLS slot pojmenovat.

   ```
   LocalDataStoreSlot slot = Thread.AllocateNamedDataSlot("jméno");
   ```

   Pojmenovaný slot je možno později najít podle jména.

   ```
   LocalDataStoreSlot slot = Thread.GetNamedDataSlot("jméno");
   ```

2. Každé vlákno při své inicializaci vytvoří lokální proměnnou (zde pro příklad typu MyClass) a do TLS uloží referenci na tento objekt.

   ```
   Thread.SetData(slot, new MyClass());
   ```

3. Při práci se svou lokální proměnnou si vlákno vytáhne referenci na svůj lokální objekt z TLS. Pokud s ní bude pracovat déle, je vhodné si ji uložit do běžné lokální proměnné metody (tj. na zásobník, mimo TLS).

   ```
   MyClass objekt = (MyClass)Thread.GetData(slot);
   ```

4. Po skončení všech pracovních vláken pak hlavní vlákno uvolní pojmenovaný TLS slot. Nepojmenované TLS sloty se v .NETu neuvolňují (uvolňuje se tedy de facto jen jméno).

```
Thread.FreeNamedDataSlot(slot);
```

B.4 Podpora TLS v překladačích jazyků

Některé překladače programovacích jazyků podporují TLS přímo, tj. na úrovni syntaxe. Používání TLS na úrovni programovacího jazyka je od výše uvedených postupů velmi odlišné. Výhodou je pohodlnější práce, přehlednější kód a silnější typová kontrola. Obecně lze říci, že TLS podporované překladači jazyků je z hlediska programátora lepší, než TLS pomocí funkcí operačního systému, protože je syntakticky jednodušší.

B.4.1 Visual C/C++

Visual C++ umožňuje označit jakoukoliv statickou a konstantně inicializovanou proměnnou modifikátorem **`__declspec(thread)`**, například takto:

```
static __declspec(thread) int lokalni;
```

Takto deklarovaná proměnná je překladačem umístěna do TLS, není třeba volat žádné další funkce operačního systému. Jelikož pracujeme v systému Windows, stále platí omezení velikosti TLS paměti uvedené v kapitole B.3.1 na straně 64.

Práce s TLS tímto způsobem zajišťuje silnou typovou kontrolu – nepracujeme již s obecnými pointery `LPVOID` (`void*`), což je jistě výhodou. Při časté práci s některou proměnnou je opět vhodné udělat is její kopii do lokální proměnné metody/funkce (tj. mimo TLS).

B.4.2 GNU C/C++ (GCC) a Sun Studio C/C++

V Linuxu a obecně mimo Windows je nejčastěji používán překladač GCC. Ten podporuje stejnou funkcionalitu jako Visual C++, ovšem s lehce jiným způsobem deklarace – používáme modifikátor `__thread`.

```
static __thread int lokalni;
```

Funkcionalita je pak stejná jako ve Visual C++. Možnost používat TLS tímto způsobem je jen na určitých mikroprocesorech (Intel x86/IA-32 a IA-64 od verze GCC 3.3). Stejnou syntaxi podporuje i Sun Studio C++ (pro Solaris a Linux).

B.4.3 Jazyky v prostředí .NET Framework

Také v prostředí .NETu lze TLS používat deklarativně. Proměnné, které chceme umístit do TLS označíme atributem `ThreadStatic`, čili
v C# např. `[ThreadStatic] static int a;`
ve Visual Basicu např. `<ThreadStatic> Shared value As Integer`

U tohoto deklarativního způsobu použití TLS se nedoporučuje používat přiřazení hodnoty přímo v definici proměnné, neboť tato inicializace by byla provedena jen jedním vláknem (prvním, které by se tomuto kódu dostalo). Toto omezení je samozřejmě analogické chování jazyka C++.

B.5 Případová studie: BiF

V předchozích kapitolách bylo popsáno několik způsobů použití TLS. Pro názornost uvedeme praktický příklad -- kód, který používá TLS se syntaktickou podporou Visual C++.

B.5.1 Co je to BiF

BiF je binární faktorizační program sloužící ke statistické analýze dat pomocí metody zvané binární faktorová analýza. Tato metoda je v programu BiF implementována několika různými algoritmy, z nichž pro většinu úloh je nejvhodnější GABFA, výpočet založený na modifikovaném genetickém algoritmu. Profilováním programu lze vysledovat, že zhruba 90% času program stráví ohodnocováním jedinců umělé populace pomocí pseudo–dělení binárních matic. Je také důležité, že

- ohodnocení jednoho jedince populace sestává z 99% z jednoho maticového dělení
- během výpočtu se ohodnocuje velké množství jedinců
- každé ohodnocení jednoho jedince je nezávislé na ostatních
- pamatujeme si nejlepšího jedince, kterého během výpočtu najdeme

B.5.2 Paralelizace

Je tedy zřejmé, že máme-li k dispozici víceprocesorový počítač (obvykle dvouprocesorový nebo dvoujádrový), největšího zrychlení dosáhneme, právě když se zaměříme na paralelizaci pseudo–dělení binárních matic nebo ohodnocování jedinců populace. Druhá varianta se ukazuje jako

mnohem snazší – jelikož jednotliví jedinci v populaci jsou na sobě navzájem nezávislí, můžeme dělení provádět paralelně v tolika vláknech, kolik máme fyzických procesorů či jader.

TLS pak využijeme právě ve třídě provádějící dělení matic. Paralelizaci této operace můžeme samozřejmě provést i bez TLS způsobem popsaným v kapitole B.2 na straně 63, tj. vytvořením instance této třídy pro každé vlákno. Je zde však několik důvodů, proč je použití TLS lepší variantou:

- Jedná se o třídu algoritmu. Takové třídy je vhodnější vytvářet jako statické. Nepotřebujeme pak zakládat instance této třídy.
- Kód statické třídy je vždy rychlejší než kód běžné třídy (překladač má k dispozici jeden registr procesoru navíc pro optimalizace kódu a všechny proměnné třídy jsou adresovatelné přímo bez odkazování pomocí `this`).
- Operace dělení v programu BiF používá jako dělenec (největší matici) statická konstantní data. Je vhodné tato data mít jako statickou položku dělicí třídy.

Dělicí třídu tedy v zájmu efektivity kódu deklarujeme jako statickou (tj. všechny součásti třídy jsou statické). Aby statická třída byla použitelná i pro vícevláknové programy, všechny lokální proměnné používané při výpočtu musí být buď lokální v rámci metody (což platí ve většině případů), nebo lokální v rámci vlákna (čili umístěné na TLS).

V našem případě umístíme na TLS především matici nejlepšího jedince, kterého během výpočtu najdeme (toto zapamatování je požadováno algoritmem, viz výše). Matice nejlepšího jedince je uložena v běžném poli. Používali-li bychom jen jedno pole pro všechna vlákna, museli bychom během výpočtu provádět synchronizaci vláken pomocí kritické sekce, aby byl zaručen konzistentní stav tohoto pole. Časté používání kritických sekcí zpomaluje vlastní výpočet. Alokujeme-li pro každé vlákno samostatné pole pro uložení matice nejlepšího jedince, vícevláknový výpočet může běžet zcela bez vzájemné synchronizace vláken. Do společné paměti ukládáme jen údaj kvality nejlepšího řešení (což je celé číslo – int) a to lze jednoduše implementovat pomocí tzv. interlocked operací (tj. bez kritických sekcí).

S použitím TLS jsme tedy dosáhli toho, že dělicí algoritmus je ve statické třídě (máme přehlednější a rychlejší kód) a výpočet navíc běží bez vzájemné synchronizace vláken (máme kratší a rychlejší kód). S přímou podporou TLS v překladači C++ je kód i velmi přehledný a je využita silná typová kontrola jazyka (samozřejmě jen do míry typové kontroly v C++).

Shrnutí

Představili jsme si Thread Local Storage a jeho použití v různých systémech, platformách i jazycích. Na praktickém příkladě jeho použití bylo demonstrováno, jaké výhody může TLS přinést při optimalizaci kódu pro počítače s více procesory nebo vícejádrovými procesory, které začínají být dnes již naprosto běžné.

Technologie TLS úzce souvisí s objektově–orientovaným programováním, neboť obojí je vhodné pro každodenní programování. TLS přitom zasahuje do principu objektově–orientovaného programování (OOP), neboť deklarace proměnných jako vláknově–lokálních není v běžné teorii OOP diskutována a její opis jinými prostředky čistě na bázi OOP je nevýhodný z hlediska optimalizace kódu pro rychlost. Kód využívající TLS je kratší, přehlednější i rychlejší, než kód stejné funkcionality bez TLS.

Kontrolní otázky

1. *Jak lze lokální paměť vláken nahradit využitím kontextu?*
2. *TLS je především funkce operačního systému či platformy. Jaký ale pak má smysl či přínos speciální podpora TLS v překladačích vyšších jazyků?*

Reference

[Bar05] Blaise Barney. *POSIX Threads Programming.* Livermore Computing, 2005. http://www.llnl.gov/computing/tutorials/pthreads/

[Doe03] Doug Doedens. *Use Thread Local Storage to Pass Thread Specific Data.* C# Corner, 2003. http://www.c-sharpcorner.com/Code/2003/March/UseThreadLocals.asp

[Duf06] Joe Duffy. *Professional .NET Framework 2.0.* Wrox Press, **2006**. ISBN 0-7645-7135-4, ISBN-13: 978-0-7645-7135-0.

[Duf07] Joe Duffy. *Why the CLR 2.0 SP1's threadpool default max thread count was increased to 250/CPU.* Weblog. http://www.bluebytesoftware.com/blog/PermaLink,guid,ca22a5a8-a3c9-4ee8-9b41-667dbd7d2108.aspx

[Kep05] Aleš Keprt. Thread Local Storage. Ve sborníku konference: *Objekty 2005.* VŠB Technická Univerzita, Ostrava, **2005**, pp. 85–91. ISBN 80-248-0595-2.

[Kep06] Aleš Keprt. Kombinace C++ a .NET – jak a proč. Ve sborníku konference: *Objekty 2006.* Česká zemědělská univerzita, Praha, 2006, pp. 193–208. ISBN 80-213-1568-7.

[Kep07] Aleš Keprt. *Operační systémy.* Univerzita Palackého, **2007**. Studijní text pro distanční vzdělávání, dostupný studentům na adrese http://www.keprt.cz/vyuka/.

[MSDN] Visual Studio 2005/2008 – nápověda MSDN. Totéž je i na webu na adrese http://msdn.microsoft.com/.

[Ric00] Jeffrey Richter. Garbage Collection – Part 2: Automatic Memory Management in the Microsoft .NET Framework. V časopise: *MSDN Magazine.* Prosinec **2000**. http://msdn.microsoft.com/msdnmag/issues/1200/GCI2/

[Tro07] Andrew Troelsen. *Pro C# 2008 and the .NET 3.5 Platform, 4.vydání.* Apress, **2007**. 1370pp., ISBN 1-59059-884-9, 978-1-59059-884-9.

[Wiki] *Wikipedia, the free encyclopedia.* Internetová encyklopedie. http://www.wikipedia.org/