Platforma .NET

4. seminář

Radek Janoštík

Univerzita Palackého v Olomouci

16. 10. 2025

Reakce na úkoly

- Největší problémy IsPowOf
- Místy menší množství testů (specifické vstupy)
- Je `null` instancí `int[]` ?
- Zkoušel někdo TDD?

Paralelní programování v .NET

- Způsob obalení systémových vláken
- Vyšší výkon, neblokující GUI, vytížení HW
- 4 přístupy (delegáti, System.Threading, TPL, `async, await`)
- Některé už od .NET 1.0, jiné později
- Dostupná řada synchronizačních mechanismů (monitor, semafor, . . . (viz namespace `System.Threading`))

Delegáti – opakování

- Objektové, typově-bezpečné, zapouzdření ukazatele na metodu
- Mechanismus pro předávání metod jako parametr metod
- Při definici: `public delegate string SomeOp(string  s, int i);` kompilátor dynamicky vytvoří třídu:

```
public sealed class SomeOp : System.MulticastDelegate {
   public SomeOp(object target, uint functionAddress);
   public int Invoke(string s, int i);
   public IAsyncResult BeginInvoke(string s, int i,
   AsyncCallback cb, object state);
   public string EndInvoke(IAsyncResult result);
}
```

- Doposud jsme používali jen synchronní `string Invoke(string s, int i)` (ukázka)

Delegáti – Asynchronní povaha

- Zbylé metody nám umožní spustit kód v dalším vlákně
- Bohužel na „novém .NET“ (.NET Core a novější) nemusí fungovat
- Parametry `BeginInvoke` závisí na předpisu delegáta + přidány parametry `AsyncCallback` a `object` (ukázka později)
- Návratovou hodnotou je `IAsyncResult`
- Návratovou hodnotou `EndInvoke` je int (opět podle předpisu delegáta), parametr `IAsyncResult` (korespondence)

Asynchronní zavolání metody

- Kód můžeme asynchronně spustit pomocí `IAsyncResult res = op.BeginInvoke("Some string", 3, null, null);`
- Kód se automaticky pustí v (nějakém) ”novém” vlákně
- Pak můžeme dělat něco jiného v původním vlákně
- Počkání na dokončení pomocí `op.EndInvoke(res)`

Aktivní čekání

- `IAsyncResult` má vlastnost `bool IsCompleted`
- Můžeme aktivně počkat na výsledek, případně udělat další práci

```
IAsyncResult res = op.BeginInvoke("Async volání", 5, null, null);
while (!res.IsCompleted) {            {
   Console.WriteLine("Tady něco dělám");
   Thread.Sleep(500);
}
string vysledek = op.EndInvoke(res); // čekat se už nebude
Console.WriteLine(vysledek);
```

- Případně pomocí `AsyncWaitHandle` (ukázka)

AsyncCallback delegáta

- Místo aktivního čekání, zda už proces skončil, můžeme využít AsyncCallback
- Metoda bude zavolána po dokončení kódu delegáta; volána ve **vlákně volaného**!
- Callback musí splňovat `void MyAsyncCallbackMethod(IAsyncResult iar)` (ukázka)
- Poslední parametr `BeginInvoke` je `object`, slouží k předání informací do callbacku

System.Threading

- Tento postup již známe z minula – opakování
- `Thread.CurrentThread` – aktuální vlákno
  - `CurrentCulture` a `CurrentUICulture`
  - Základní vlastnosti

IsAlive	true	bool
IsBackground	false	bool
IsThreadPoolThread	false	bool
ManagedThreadId	9	int
Name	"Vlákno"	string
Priority	Normal	System.Threading.ThreadPriority
ThreadState	Running	System.Threading.ThreadState

- `ThreadPriority` má hodnoty `Highest, AboveNormal, Normal, BelowNormal, Lowest`
- `ThreadState` má hodnoty `Aborted,AbortRequested, Background, Running, Stopped, StopRequested, Suspended, SuspendRequested, Unstarted, WaitSleepJoin`

Vytvoření nového vlákna

- Konstruktor má parametr delegáta se signaturou

```
public delegate void ThreadStart();
Thread worker = new Thread(RunFactorial);
```

- Spuštění vlákna `worker.Start()`
- Předání parametrů

```
public delegate void ParametrizedThreadStart(object o);
worker.Start(param);
```

- Nebo

```
Thread t = new Thread((ThreadStart) delegate
{Console.WriteLine(Fac(7));});
t.Start();
```

Manipulace s vlákny

- Pozastavení činnosti vlákna – `thread.Suspend();`
- Opětovné spuštění – `thread.Resume();`
- Zastavení činnosti – `thread.Abort();` – využití výjimek
- Ke změně může dojít se zpožděním
- Počkání na dokončení činnosti – `thread.Join();`

Synchronizace

- Při přístupu ke sdíleným proměnným více vlákny může dojít k nechtěnému stavu
- Problém atomicity operací: $x = x + 5$ vnitřně není jedna operace
- Příkaz `lock (x) { }` zapouzdří objekt x do kritické sekce (vzájemné vyloučení)
- `lock` je jen syntaktický cukr za použití `System.Threading.Monitor` (viz dokumentace)
- Pozor na deadlock (zablokování) a zamykání stringů

Atomické operace

- Předchozí chyby souběhu způsobeny neatomicitou operací
- `System.Threading.Interlocked` poskytuje atomické operace s nižší režií než `lock`
  - `CompareExchange()` – 3 argumenty – porovná dva a v případě rovnosti nahradí první třetím
  - `Decrement()` – „atomická“ (= bezpečná) dekrementace
  - `Increment()` – bezpečná inkrementace
  - `Exchange()` – bezpečně vymění dvě hodnoty

Thread-save pro líné

- Atribut `[Synchronization]` z namespace `System.Runtime.Remoting.Contexts`
- Zamkne **všechny** membery dané třídy
- Třída musí dědit z `ContextBoundObject`
- Výhoda – rychlý vývoj
- Nevýhoda – k zamykání dochází vždy, i když nehrozí souběh – výkon
- Nevýhoda – fuguje jen na „starém“ .NET Frameworku

CLR ThreadPool

- Např. při asynchronním volání delegátů se vždy nevytváří nové vlákno (drahá operace)
- CLR si udržuje pool vláken, které může recyklovat

```
WaitCallback workToDo = new WaitCallback(doSomething);
for (int i = 0; i < 4; i++) {
   ThreadPool.QueueUserWorkItem(workToDo, null);
}
```

- Výhoda – minimalizuje se počet (přímo) vytvořených vláken
- Nevýhody – Nemůžeme pozastavit, zastavit (neznáme jméno), vždy běží s normální prioritou

Task Parallel Library

- Od .NET 4.0 (Třída `System.Threading.Tasks`
- Umožňuje některé věci dělat paralelně, aniž bychom (přímo) používali vlákna nebo `ThreadPool`
- Automatická distribuce zátěže na dostupná jádra (odstiňuje od nízkoúrovňových věcí)
- Hlavní třídy: `Parallel` a `Task`
- Pořád musíme myslet na to, zda se vyplatí práci dělat paralelně

Datový paralelismus

- Potřeba něco udělat s každým prvkem nějaké kolekce
- Metody `Parallel.For` a `Parallel.ForEach`
- Definice `For(Int32, Int32, Action<Int32>)`

```
Parallel.For(0, 100, (i) =>
            {
                pole[i] = i * i * i; // dlouhý vypocet
                Thread.Sleep(100);
            });
```

- Podobně ForEach

Zrušení Parallel.ForEach

- Často je potřeba přerušit práci v nějakém vlákně (GUI)
- Zrušení výpočtů pomocí `CancelationToken` předaný jako parametr v `ParallelOptions`
  - `ParallelOptions` může obsahovat také maximální počet souběžných úloh
  - A také `TaskScheduler` – `https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskscheduler` – doporučuji pročíst

```
CancellationTokenSource tokenSource = new CancellationTokenSource();
ParallelOptions opts = new ParallelOptions();
opts.MaxDegreeOfParallelism = 8;
opts.CancellationToken = tokenSource.Token;

Parallel.ForEach(velkePole, opts, integer => {});
```

- Viz ukázka kódu

Task paralelismus

- Spuštění více paralelních úloh pomocí `Parallel.Invoke()`

```
Parallel.Invoke(
 () => {
     Thread.Sleep(500); // nějaký výpočet
     Console.WriteLine($"Nacházíme se v vlákně s ID: {Thread.CurrentThread.ManagedThreadId}.");
    },
 () => {
     Thread.Sleep(300); // nějaký výpočet
     Console.WriteLine($"Další výpočet ve vlákně s ID: {Thread.CurrentThread.ManagedThreadId}.");
    },
 () => {
     Thread.Sleep(600); // nějaký výpočet
     Console.WriteLine($"třetí výpočet ve vlákně s ID: {Thread.CurrentThread.ManagedThreadId}.");
    });
Console.WriteLine("Hotovo!");
```

- Počká, až každá operace dokončí svou práci

PLINQ

- Možnost dělat LINQ dotazy paralelně
- CLR odhadne, zda je lepší to dělat klasicky nebo paralelně
- Opět možné zrušení (podobně jako výše)
- Ilustrační příklad:

```
int[] velkePole = new int[100000000]; // + naplnit
List<int> sedmi = velkePole.Where(p => p % 7 == 0).ToList(); // klasicky

List<int> sedmiP = velkePole.AsParallel().WithDegreeOfParallelism(8)
                                          .Where(p => p % 32323 == 0).ToList();
```

Klíčová slova async a await

- Nová klíčová slova `async` a `await` od .NET 4.5
- `async` označuje metodu, že může být automaticky volána v jiném vlákně
  - CLR při volání automaticky zařídí vytvoření vlákna a spuštění metody
- `await` se dává před volání metody
  - Pozastaví současné vlákno a čeká na dokončení volané metody
  - Po dokončení pokračuje dále
- `async` metody obalují návratový typ do `Task<typ>`
- Volání z non-async metody: CreateSomeStringAsync().Result
- Pěkný příklad `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/` (samostudium)

Úkol (0/3) – reader writer problem

- Pomocí Thread a synchronizačního primitiva semafor implementujte problém čtenáře a písaře
- Do sdílené paměti čtenář může zapisovat pouze jeden písař, ale číst z paměti může více čtenářů zároveň
- Řešení nemusí být férové

Úkol (1/3)

- Programově stáhnout datasety z UC Irvine Machine Learning Repository:
  - `https://archive.ics.uci.edu/ml/machine-learning-databases/flags/flag.data` (Flags)
  - `https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/breast-cancer-wisconsin.data` (brest-cancer)
  - `https://archive.ics.uci.edu/ml/machine-learning-databases/mushroom/agaricus-lepiota.data` (mushroom)
  - `https://archive.ics.uci.edu/ml/machine-learning-databases/zoo/zoo.data` (zoo)
  - `https://archive.ics.uci.edu/ml/machine-learning-databases/voting-records/house-votes-84.data` (house votes)

Úkol (2/3)

- Pro každý dataset zjistit:
  - Jeho velikost (kB)
  - Počet řádků a sloupců
  - Počet znaků
  - Pro každý znak v datasetu počet jeho výskytů
- Nejprve vše udělat sekvenčně
- Poté zvolit vhodnou metodu paralelizace a co půjde, to zparalelizovat
- Porovnat dobu trvání obou postupů

Úkol (3/3) – ukázka

```
Stahuji datasety sekvenčně!
Zjišťuji informace!
Dataset xyz má takové vlastnosti (výpis statistik)
Dataset xzy má takové vlastnosti (výpis statistik)
...
Sekvenční doba: XYs

Stahuji datasety paralelně!
...
...
Paralelně to trvalo: YXs
```