Microsoft

Vladimír Juhás

Vývoj paralelnej aplikácie v jazyku C# 4.0

OD študentov PRE študentov

Fakulta hospodárskej informatiky EU v Bratislave

Microsoft Visual Studio 2010

Autor: Ing. Vladimír Juhás
Odborný garant: Ing. Ján Hanák, PhD., MVP

Vývoj paralelnej aplikácie v jazyku C# 4.0
Praktické cvičenie zo série „Od študentov pre študentov"

Charakteristika praktických cvičení zo série „Od študentov pre študentov"

Sme presvedčení o tom, že keď inteligentní mladí ľudia ovládnu najmodernejšie počítačové technológie súčasnosti, ich tvorivý potenciál je vskutku nekonečný. Primárnym cieľom iniciatívy, ktorá stojí za sériou praktických cvičení „Od študentov pre študentov", je maximalizácia hodnôt ľudských kapitálov študentov ako hlavných členov akademických komunít. Praktické cvičenia zo série „Od študentov pre študentov" umožňujú študentom využiť ich existujúce teoretické znalosti, pričom efektívnym spôsobom predvádzajú, ako možno tieto znalosti s výhodou uplatniť pri vývoji atraktívnych počítačových aplikácií v rôznych programovacích jazykoch (C, C++, C++/CLI, C#, Visual Basic, F#). Budeme nesmierne šťastní, keď sa našim praktickým cvičeniam podarí u študentov prebudiť a naplno rozvinúť záujem o programovanie a vývoj počítačového softvéru. Veríme, že čím viac sofistikovaných IT odborníkov vychováme, tým viac budeme môcť spoločnými silami urobiť všetko pre to, aby sa z tohto sveta stalo lepšie miesto pre život.

Vývoj paralelnej aplikácie v jazyku C# 4.0

Cieľové publikum: **študenti** s pokročilou proficienciou v jazyku C#
Vedomostná náročnosť: ■■■□□
Časová náročnosť: **1** hodina a **30** minút
Programovacie jazyky: **C#** (vo verzii 4.0)
Vývojové prostredia: **Visual Studio 2010**
Operačné systémy: **Windows 7**, Windows Vista, Windows XP
Technológie: **bázová knižnica jazyka C#**

Softvérový priemysel v týchto dňoch stojí opäť na významnej križovatke. Výrobcovia hardvérových komponentov narazili pri vývoji nových mikroprocesorov na fyzikálne limity v oblasti zvyšovania pracovnej frekvencie a s tým súvisiacu energetickú náročnosť ich chladenia. Vývoj nových mikroprocesorov sa preto začal uberať cestou zmenšovania výrobného procesu, ktorý umožňuje na plochu jedného mikroprocesora umiestniť čoraz väčší počet exekučných jadier. V tejto súvislosti sa experti z oblasti počítačových vied zhodujú v tom, že prichádza tzv. viacjadrová revolúcia, ktorá so sebou prináša možnosti zvýšenia výkonnosti existujúcich algoritmov a vytvárania úplne nových paralelných algoritmov. Máme teda príležitosť vytvárať algoritmy, ktoré sa môžu stať oveľa užitočnejšími a zaujímavejšími pre svojich používateľov.

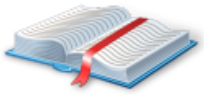
Viacjadrová revolúcia prináša výrazné zmeny v prístupe k tvorbe počítačového softvéru. Nie je možné sa ďalej spoliehať na nárast výkonnosti svojich algoritmov, „len" z dôvodu zvýšenia pracovnej frekvencie procesorov inštalovaných na serveroch alebo pracovných staniciach koncových používateľov. Vývojári musia zmeniť filozofiu svojho myslenia, ak chcú využiť možnosti, ktoré im viacjadrová revolúcia prináša. Musia začať chápať a vytvárať aplikácie nie ako sekvenčnú postupnosť krokov alebo inštrukcií, ale ako súbor úloh, ktoré majú byť paralelne vykonané.

Príchod nových hardvérových technológií v podobe mikroprocesorov s viacerými exekučnými jadrami spôsobil, že popredný výrobcovia nástrojov na tvorbu počítačového softvéru postupne uvádzajú na trh nové verzie svojich vývojových prostredí, programovacích jazykov a technológií zameraných na paralelizáciu už existujúcich sekvenčných aplikácií, prípadne na vytváranie úplne nových paralelných aplikácií. Vývoj v tomto smere pokročil už tak ďaleko, že vysoko sofistikované technológie z oblasti vedy a výskumu sa stávajú každodennou súčasťou života komerčných vývojárov. Nové nástroje umožňujú efektívny vývoj, testovanie a profilovanie paralelných aplikácií schopných plne využiť dostupný výpočtový výkon nových generácií mikroprocesorov.

Na vývojárov tu však čaká množstvo metód a techník pri uplatňovaní novej filozofie vývoja počítačového softvéru akou je paralelné objektovo orientované programovanie (POOP), ktoré so sebou môže do aplikácií priniesť okrem mnohých pozitív aj nové druhy vážnych chýb. Jednou z nich je napríklad nedeterminizmus vo výstupoch paralelných aplikácií. Napriek tomu však pozitíva, ktoré so sebou POOP podporené novými technológiami prináša, vysoko prevažujú nad jeho negatívami.

Praktické cvičenie je rozdelené na dve časti. Cieľom prvej časti je predstaviť možnosti vývojovo–exekučnej platformy Microsoft .NET Framework 4.0 a jazyka C# 4.0 z pohľadu využitia paradigmy POOP. Druhá časť si kladie za cieľ využiť nadobudnuté teoretické poznatky pri vývoji paralelnej

aplikácie zameranej na odhaľovanie šifrovaných prístupových hesiel do informačných systémov. Aplikácia implementuje a zároveň porovnáva rôzne prístupy k paralelizácii svojich algoritmov.



Obsah praktického cvičenia

1 Paralelné objektovo orientované programovanie (POOP) v jazyku C# 4.0

Medzi najdôležitejšie inovácie vývojovo–exekučnej platformy Microsoft .NET Framework 4.0 v oblasti paralelného objektovo orientovaného programovania (**POOP**) patrí predovšetkým knižnica **Task Parallel Library (TPL)** a paralelná verzia programových konštrukcií vstavaného dopytovacieho jazyka Language Integrated Query to Objects (LINQ to Objects) teda **Parallel LINQ (PLINQ)**. Ďalšou novinkou je **uvedenie špeciálnych vláknovo bezpečných dátových štruktúr** podporujúcich paralelizáciu programových inštrukcií. Uvedené novinky umožňujú vývojárom prácu na skutočne vysokej úrovni abstrakcie od hardvéru so značným vplyvom na zvýšenie ich pracovnej produktivity.

1.1 Programové vlákna

Programové vlákno v jazyku C# 4.0 predstavuje inštancia triedy **Thread**. Táto trieda sa nachádza v mennom priestore **System.Threading**. Inštancie triedy **Thread** sú v paralelných programoch priamo asociované s natívnymi vláknami operačného systému. Tieto vlákna sú ďalej v hierarchii vlákien mapované na hardvérové vlákna, ktoré spravuje samotný procesor. V prípade hardvérových vlákien platí, že jedno exekučné jadro procesora je schopné v istom časovom okamihu spracovať iba inštrukcie jedného hardvérového vlákna. Z pohľadu abstrakcie od hardvéru operujú programové vlákna na najvyššej úrovni abstrakcie, vlákna operačného systému na nižšej úrovni a hardvérové vlákna sa vyznačujú najnižšou úrovňou abstrakcie od hardvéru.

Ak chce programátor dosiahnuť maximálne využitie dostupného výpočtového výkonu hardvérovej platformy, na ktorej bude paralelná aplikácia spúšťaná, musí vo vlastnej réžii zabezpečiť všetky predpoklady paralelnej exekúcie viacvláknovej riadenej aplikácie[1]:

- **Optimálny počet programových vlákien** – predstavuje počet exekučných jadier viacjadrového procesora. Tento predpoklad zabezpečí mapovanie každého programového vlákna na jedno exekučné jadro procesora. V praxi sa môžeme stretnúť ešte s dvomi neoptimálnymi variantmi. Prvým variantom je ten, pri ktorom je počet vlákien aplikácie menší ako počet exekučných jadier procesora. Tento variant nedokáže zabezpečiť maximálnu efektivitu využitia výpočtových zdrojov počítačového systému. V druhom variante má aplikácia väčší počet vlákien ako počet exekučných jadier procesora. V tomto prípade ide o špeciálny exekučný model kde $\boldsymbol{n} > \boldsymbol{E_x}$, $\boldsymbol{n}$ predstavuje počet programových vlákien a $\boldsymbol{E_x}$ počet exekučných jadier procesora. Programové vlákna sú potom rozdelené do tzv. súprav programových vlákien, každá súprava programových vlákien je distribuovaná na iné exekučné jadro procesora. Exekúcia súprav programových vlákien je navzájom paralelná, ale exekúcia programových vlákien vo vnútri každej súpravy prebieha pseudoparalelným spôsobom.

[1] HANÁK, J. 2009. *Základy paralelného programovania v jazyku C# 3.0.* Brno: Artax, 2009. 201 s. ISBN 978-80-87017-03-6.

- **Škálovateľnosť programových vlákien** – predstavuje schopnosť viacvláknovej riadenej aplikácie flexibilne sa prispôsobiť hardvérovej platforme, na ktorej je práve exekvovaná. To znamená, že aplikácia dokáže pre danú hardvérovú platformu vytvoriť optimálny počet exekučných vlákien.

- **Predpoklad optimálnej distribúcie pracovného zaťaženia programových vlákien –** je splnený v prípade, že každé z $\boldsymbol{n}$ programových vlákien spracováva $\frac{\mathbf{1}}{\boldsymbol{n}}$ zo všetkých programových inštrukcií danej viacvláknovej riadenej aplikácie.

- **Eliminácia závislostí medzi viacerými programovými vláknami** – v praxi je zabezpečenie bezpečného konkurenčného prístupu k zdieľaným zdrojom z rôznych programových vlákien často veľmi náročná úloha. Splnenie tohto predpokladu je možné zabezpečiť dvomi spôsobmi. Vylúčením zdieľaných zdrojov, teda tzv. privatizáciou zdrojov. Tento spôsob spočíva v poskytnutí kópie zdieľaného objektu každému programovému vláknu, ktoré s týmto objektom bude pracovať. Vedľajším efektom tohto spôsobu je vyššia pamäťová náročnosť aplikácie. Druhým spôsobom je zabezpečenie synchronizovaného prístupu k zdieľanému zdroju z rôznych programových vlákien. Nežiaducim efektom je v tomto prípade prechod k sekvenčnej exekúcii počas synchronizácie prístupu k zdieľanému zdroju.

Na tomto mieste je nutné podotknúť, že vo všeobecnosti sa odporúča vytvárať maximálne taký počet programových vlákien, aby bolo každé vlákno paralelnej aplikácie namapované na práve jedno exekučné jadro procesora. Toto pravidlo je potrebné dodržiavať z dôvodu efektívneho využívania systémových zdrojov, pretože každé ďalšie programové vlákno na seba viaže postrannú réžiu, v podobe nutnosti jeho správy spoločným behovým prostredím (CLR). Ďalším z dôvodov je, že s počtom programových vlákien narastá náročnosť udržiavania a čitateľnosti programového kódu. V neposlednom rade ide o možnosť vzniku konfliktov a pretekov medzi vláknami súperiacimi o zdieľané zdroje. Ide najmä o preteky vlákien (race conditions) a uviaznutia (deadlocks). K zdieľaným zdrojom patria:

- Systémové zdroje, napríklad komunikačné porty.
- Zdroje zdieľané viacerými procesmi, napríklad súbory.
- Zdroje v rámci jednej aplikačnej domény, napríklad globálne, statické a inštančné premenné.

Naopak, model paralelizmu založený na vláknach sa stáva veľmi výhodným riešením, ak zámerom vývojára paralelnej aplikácie je:

- Vykonávať časť programu paralelne, pričom predpokladá, že jej exekúcia bude časovo náročná.
- Disponovať možnosťou zmeny priority exekúcie tej časti programu, ktorá má byť exekvovaná paralelne.
- Zabezpečiť odozvu používateľského rozhrania aj počas vykonávania operácií náročných na výpočtový výkon.

1.1.1 Inštanciácia triedy Thread

Triedu **Thread**, ktorej zrodené inštancie predstavujú pracovné programové vlákna, môžeme najjednoduchším spôsobom inštanciovať pomocou nasledujúceho príkazu:

```
Thread programoveVlakno = new Thread(Metoda);
```

Po exekúcii tohto príkazu bude odkazová premenná **programoveVlakno**, ktorej typom je trieda **Thread**, uchovávať odkaz na inštanciu tejto triedy nachádzajúcu sa na riadenej halde. Uvedený inštanciačný príkaz však nezobrazuje fakt, že pri volaní parametrického inštančného konštruktora triedy **Thread** s nasledujúcou signatúrou, dochádza aj k inštanciácii delegáta **ThreadStart**.

```
public Thread(ThreadStart start) { ... }
```

Konštruktoru je poskytnutý odkaz na bezparametrickú metódu bez návratovej hodnoty s identifikátorom **Metoda**. Táto metóda bude neskôr podrobená exekúcii na pracovnom programovom vlákne.

Syntaktickou obmenou prvého inštanciačného príkazu bez zmeny jeho funkcionality je príkaz:

```
Thread programoveVlakno = new Thread(new ThreadStart(Metoda));
```

Ďalšou možnou variáciou inštanciačného príkazu, ktorá nemení jeho funkcionalitu, je príkaz, ktorý parametrickému inštančnému konštruktoru triedy **Thread** odovzdáva pomocou λ-výrazu odkaz na kompilátorom automaticky vytvorenú bezparametrickú anonymnú metódu, v ktorej tele sa nachádza volanie metódy **Metoda**. Je zrejmé, že v tele kompilátorom vytvorenej anonymnej metódy sa môže nachádzať okrem volania metódy **Metoda** ľubovoľná množina programových príkazov.

```
Thread programoveVlakno = new Thread(() =>
{
      Metoda();
});
```

Parametrický inštančný konštruktor triedy **Thread** vo svojej preťaženej verzii dovoľuje pracovať s parametrickým delegátom **ParametrizedThreadStart**, ktorého inštancia v sebe môže uchovávať odkaz na parametrickú metódu. Jediným parametrom tejto metódy musí byť odkazová premenná typu **object**.

```
public Thread(ParametrizedThreadStart start) { ... }
```

Inštanciačný príkaz, pri ktorého exekúcii je volaný konštruktor s touto signatúrou, musí spĺňať podmienku, aby v ňom použitá metóda deklarovala jeden formálny parameter typu **object**. Následne môžeme inštanciačný príkaz zapísať v tvare:

```
Thread programoveVlakno = new Thread(new ParametrizedThreadStart (Metoda));
```

V prípade, že vývojár nechce explicitne definovať parametrickú metódu bez návratovej hodnoty, môže inštanciovať programové vlákno pomocou λ–výrazu, pri ktorom je kompilátorom automaticky vytvorená anonymná metóda s jedným formálnym parametrom **parameter**, ktorého typom je **object**.

```
Thread programoveVlakno = new Thread((object parameter) =>
{
        // Príkazy v tele automaticky vytvorenej anonymnej metódy.
});
```

1.1.2 Vybrané metódy triedy Thread a ich použitie

Po úspešnom vytvorení inštancie triedy **Thread** ma vývojár k dispozícii pracovné programové vlákno pripravené na spustenie asynchrónnej exekúcie programových príkazov s ním asociovaných. Za predpokladu, že sme si vopred vytvorili inštanciu programového vlákna, ktorá pracuje s delegátom **ThreadStart**, môžeme exekúciu na tomto programovom vlákne spustiť volaním inštančnej bezparametrickej metódy **Start** triedy **Thread**.

```
programoveVlakno.Start();
```

V opačnom prípade, teda pri práci s programovým vláknom, ktoré je asociované s inštanciou delegáta **ParametrizedThreadStart**, spúšťame exekúciu volaním inštančnej parametrickej metódy **Start** triedy **Thread** s jedným formálnym parametrom typu **object**.

```
programoveVlakno.Start(100);
```

Ako je zrejmé z predchádzajúceho príkazu, použitie inštancie parametrického delegáta **ParametrizedThreadStart** nie je výhodné v prípade požiadavky zachovania typovej bezpečnosti, pretože metóde dosiahnuteľnej prostredníctvom tohto delegáta môže vývojár odovzdať v podobe argumentu akúkoľvek hodnotu hodnotového alebo odkazového dátového typu. Vzhľadom na to, že pracovné programové vlákno spúšťame z primárneho, prípadne z iného pracovného programového vlákna, je potrebné exekúciu na tomto vlákne zastaviť až do momentu, keď pracovné programové vlákno dokončí svoju exekúciu. Uvedené zabezpečí volanie inštančnej metódy **Join** triedy **Thread**.

```
programoveVlakno.Join();
```

Ďalšou z metód triedy **Thread** je statická metóda **Sleep**, ktorá definuje jeden formálny parameter typu **Int32**. Keďže ide o statickú metódu, je vždy volaná v súvislosti so samotnou triedou. Volanie tejto metódy sa uskutočňuje z primárneho, prípadne z pracovného programového vlákna. Argumentom takto volanej metódy je potom časový interval (udávaný v milisekundách), počas ktorého bude exekúcia na danom vlákne pozastavená.

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

Inštančná metóda **Abort** spôsobí vznik výnimky **ThreadAbortException** na pracovnom programovom vlákne, po ktorej je spustený proces ukončenia exekúcie na tomto vlákne. Volanie metódy **Abort** sa vyznačuje niekoľkými špecifikami. Jedným z nich je situácia, keď je metóda volaná v súvislosti

s pracovným programovým vláknom, ktorého exekúcia ešte nebola spustená. V tomto prípade je proces ukončenia exekúcie na programovom vlákne spustený ihneď po volaní metódy **Start**.

Medzi ďalšie metódy triedy **Thread** patria statické metódy:

- **AllocateNamedDataSlot**,
- **AllocateDataSlot**,
- **GetData**
- a **SetData.**

Tieto metódy slúžia na prácu s lokálnym vláknovým úložiskom s názvom **Thread Local Storage (TLS)**.

1.1.3 Vybrané vlastnosti triedy Thread

Trieda **Thread** disponuje niekoľkými skalárnymi inštančnými vlastnosťami, ktoré umožňujú klientskemu kódu vo veľkej miere ovplyvňovať správanie vytvorených inštancií.

Medzi významné inštančné vlastnosti triedy **Thread** patrí vlastnosť **Name**, pomocou ktorej môže vývojár po inštanciácii triedy získať, prípadne nastaviť identifikátor pracovného programového vlákna. Tento identifikátor sa potom dá vhodne použiť vo fáze ladenia paralelnej aplikácie. Každé pracovné programové vlákno, reprezentované inštanciou triedy **Thread**, disponuje skalárnou inštančnou vlastnosťou **IsBackground**. V prípade nastavenia tejto vlastnosti na logickú pravdu **true**, sa toto pracovné vlákno stáva vláknom v pozadí. To znamená, že prebiehajúca exekúcia programových inštrukcií na tomto pracovnom vlákne nebude brániť spoločnému behovému prostrediu v ukončení životného cyklu aplikácie .NET.

Východiskové nastavenie inštančnej vlastnosti **IsBackground** predstavuje hodnota logickej nepravdy **false**. V implicitnom nastavení je teda pracovné programové vlákno vláknom v popredí a exekúcia programových príkazov s ním asociovaných zabraňuje spoločnému behovému prostrediu v ukončení životného cyklu aplikácie .NET.

Z pohľadu priority prepínania exekúcie medzi viacerými programovými vláknami dovoľuje vývojovo–exekučná platforma .NET Framework 4.0 nastavovanie piatich stupňov priority plánovania exekúcie konkrétneho programového vlákna:

- Highest (najvyššia priorita),
- AboveNormal (priorita vyššia ako normálna priorita),
- Normal (normálna priorita),
- BelowNormal (priorita nižšia ako normálna priorita),
- Lowest (najnižšia priorita).

Nastavenie priority plánovania exekúcie programového vlákna je zabezpečené pomocou skalárnej inštančnej vlastnosti **Priority** triedy **Thread**. Samotný operačný systém však môže prioritu exekúcie jednotlivých vlákien podľa potreby meniť.

Jednou z ďalších vlastností triedy **Thread** je skalárna inštančná vlastnosť **ThreadState**. Pomocou tejto vlastnosti je vývojár schopný z klientskeho kódu zistiť stav, v ktorom sa dané programové vlákno v istom okamihu nachádza.

Programové vlákno sa môže nachádzať v jednom z nasledujúcich desiatich stavov:

- **Running** – exekúcia programových príkazov asociovaných s programovým vláknom bola spustená, vlákno nie je blokované a zároveň nebola generovaná výnimka **ThreadAbortException**.

- **StopRequested** – je požadované zastavenie exekúcie programového vlákna.

- **SuspendRequested** – je požadované pozastavenie exekúcie programového vlákna.

- **Background** – indikuje, či je programové vlákno vláknom v popredí, alebo vláknom v pozadí. Tento stav kontroluje skalárna inštančná vlastnosť **IsBackground**.

- **Unstarted** – programové vlákno bolo vytvorené, ale v súvislosti s ním zatiaľ nebola zavolaná metóda **Start**. Tento stav je inicializačným stavom každého programového vlákna vytvoreného spoločným behovým prostredím.

- **Stopped** – exekúcia programových príkazov na programovom vlákne je zastavená. Programové vlákno už v tomto stave nevykonáva žiadnu činnosť.

- **WaitSleepJoin** – exekúcia programových príkazov na programovom vlákne je pozastavená – blokovaná, tento stav môže byť spôsobený volaním inštančných metód **Join** alebo **Sleep**, prípadne synchronizáciou prístupu k zdieľaným zdrojom.

- **Suspended** – exekúcia programových príkazov na programovom vlákne je pozastavená – blokovaná.

- **AbortRequested** – inštančná metóda **Abort** bola zavolaná, nebola však dosiaľ generovaná výnimka **ThreadAbortException**.

- **Aborted** – exekúcia programových príkazov na programovom vlákne je zastavená, stav programového vlákna sa však ešte nezmenil na **Stopped**.

Dôležitou poznámkou je, že skalárna inštančná vlastnosť **ThreadState** by mala byť vývojármi používaná iba na účel ladenia paralelnej aplikácie a nikdy nie na účel synchronizácie aktivít programových vlákien.

1.2 Programové vlákno v pozadí – BackgroundWorker

Jedným z mnohých účelov konštrukcie pracovných programových vlákien v paralelných aplikáciách je presunutie istej množiny programových príkazov na iné pracovné programové vlákno. Spravidla ide o exekúciu súpravy programových príkazov (operácií), ktoré by časovou náročnosťou svojej exekúcie spôsobili znehybnenie používateľského rozhrania. Inými slovami, používateľské rozhranie by prestalo reagovať na povely používateľa až do momentu finalizácie spracúvaných akcií.

Príkladom časovo náročných operácií môžu byť tieto:

- Prístup k údajom uložených na pevnom disku, ktorého rýchlosť je v porovnaní s rýchlosťou prístupu k operačnej pamäti a vyrovnávacím pamätiam procesora veľmi nízka.
- Vykonávanie databázových transakcií.
- Interakcia s webovskými službami.
- Komplexné výpočty.
- Sťahovanie rôzneho obsahu z internetu a iné vstupno–výstupné operácie.

Za vhodný model riešenia tohto problému pri vývoji aplikácií s grafickým používateľským rozhraním považujeme použitie inštancie triedy **BackgroundWorker** z menného priestoru **System.ComponentModel**, ktorá umožňuje aplikácii asynchrónne vykonávať časovo náročné operácie na samostatnom pracovnom programovom vlákne. Ako pridanú hodnotu poskytuje rozhranie v podobe súboru udalostí, metód a skalárnych inštančných vlastností na vysokej úrovni abstrakcie.

1.2.1 Inštanciácia triedy BackgroundWorker

Trieda **BackgroundWorker** je potomkom triedy **Component** z menného priestoru **System.ComponentModel**, ktorá zavádza implementáciu rozhrania **IComponent** z menného priestoru **System.ComponentModel**. Táto skutočnosť umožňuje jej integráciu do vývojového prostredia Visual Studio 2010.

Integrácia do vývojového prostredia znamená, že vývojár paralelnej aplikácie môže zabezpečiť inštanciáciu triedy **BackgroundWorker** jednoduchým „potiahnutím" komponentu z okna **Toolbox** vývojového prostredia do okna vizuálneho návrhára grafického používateľského rozhrania aplikácie. To má za následok generovanie kódu inštanciačného príkazu vývojovým prostredím v tvare:

```
this.backgroundWorker1 = new System.ComponentModel.BackgroundWorker();
```

Inštanciácia triedy **BackgroundWorker** potom prebehne v tele inštančnej metódy **InitializeComponent**, ktorá je automaticky generovaná vývojovým prostredím. Táto metóda je štandardne volaná z konštruktora triedy formulára riadenej aplikácie .NET s grafickým používateľským rozhraním.

1.2.2 Vybrané udalosti triedy BackgroundWorker a ich použitie

Ďalším krokom po úspešnej inštanciácii objektu triedy **BackgroundWorker** je zaregistrovanie metód, ktoré budú exekvované v prípade vzniku asynchrónnych udalostí. Metódy pre všetky v nasledujúcom texte opísané asynchrónne udalosti je možné zaregistrovať pomocou vývojového prostredia Visual Studio 2010, a to kliknutím na inštanciu komponentu **BackgroundWorker** v okne návrhára grafického používateľského rozhrania a zobrazením okna **Properties** pre tento komponent. V okne **Properies** po kliknutí na tlačidlo **Events** môže vývojár zaregistrovať metódy pre všetky asynchrónne udalosti inštancie triedy **BackgroundWorker**.

Prvou z asynchrónnych udalostí triedy **BackgroundWorker** je udalosť **DoWork**, ktorá nastane po volaní preťaženej inštančnej metódy **RunWorkerAsync** triedy **BackgroundWorker**. Metóda bude podrobnejšie opísaná v časti podkapitoly *1.2.3 Vybrané metódy triedy BackgroundWorker a ich použitie*.

Po zaregistrovaní udalosti pomocou vývojového prostredia bude v tele metódy **InitializeComponent** generovaný nasledujúci programový príkaz:

```
this.backgroundWorker1.DoWork +=
    new System.ComponentModel.DoWorkEventHandler(this.backgroundWorker1_DoWork);
```

Príkaz inštanciuje delegáta **DoWorkEventHandler** z menného priestoru **System.ComponentModel**, ktorý odkazuje na cieľovú metódu **backgroundWorker1_DoWork**. Následne je vytvorená inštancia delegáta prepojená pomocou preťaženého operátora **+=** s udalosťou **DoWork**. Príkaz zabezpečí, že po vzniku udalosti **DoWork** budú spracované programové príkazy v tele metódy **backgroundWorker1_DoWork**, ktorá bude volaná nepriamo pomocou delegáta. V tele metódy **backgroundWorker1_DoWork** sa musia nachádzať programové príkazy, ktoré majú byť podrobené exekúcii na pracovnom programovom vlákne.

V prípade, že asynchrónne realizované programové príkazy produkujú istý výsledok svojho spracovania, tento môže byť priradený do skalárnej inštančnej vlastnosti **Result** inštancie triedy **DoWorkEventArgs**. Inštancia tejto triedy predstavuje druhý formálny parameter metódy **backgroundWorker1_DoWork** a jej obsahom sú dáta udalosti **DoWork**, medzi inými aj výsledok spracovania asynchrónnej operácie v podobe skalárnej inštančnej vlastnosti **Result**.

Asynchrónna udalosť **ProgressChanged** triedy **BackgroundWorker** nastane v prípade volania preťaženej inštančnej metódy **ReportProgress**, ktorá bude podrobnejšie opísaná v časti podkapitoly *1.2.3 Vybrané metódy triedy BackgroundWorker a ich použitie*. Po zaregistrovaní udalosti pomocou vývojového prostredia bude v tele metódy **InitializeComponent** podobne ako v prípade udalosti **DoWork** generovaný zodpovedajúci programový príkaz:

```
this.backgroundWorker1.ProgressChanged +=
    new System.ComponentModel.ProgressChangedEventHandler(
    this.backgroundWorker1_ProgressChanged);
```

Programový príkaz vytvára inštanciu delegáta **ProgressChangedEventHandler**, ktorá je spojená s cieľovou metódou **backgroundWorker1_ProgressChanged**. Vytvorená inštancia delegáta je

následne pomocou preťaženého operátora += spojená s udalosťou **ProgressChanged**. Je nutné dodať, že použitie tejto udalosti nie je nutnou podmienkou pre úspešnú implementáciu triedy **BackgroundWorker** v paralelnej aplikácii.

V tele metódy **backgroundWorker1_ProgressChanged** je možné implementovať reakciu na progres exekúcie na pracovnom programovom vlákne, a teda na progres exekúcie v metóde **backgroundWorker1_DoWork**.

Ďalšou z asynchrónnych udalostí triedy **BackgroundWorker** je udalosť **RunWorkerCompleted**, ktorá nastane po ukončení, zrušení alebo vzniku výnimky asynchrónnej operácie na pracovnom programovom vlákne. Vývojové prostredie Visual Studio 2010 aj v tomto prípade generuje potrebný programový kód do tela metódy **InitializeComponent**.

```
this.backgroundWorker1.RunWorkerCompleted +=
    new System.ComponentModel.RunWorkerCompletedEventHandler(
    this.backgroundWorker1_RunWorkerCompleted);
```

1.2.3 Vybrané metódy triedy BackgroundWorker a ich použitie

Preťažená inštančná metóda **RunWorkerAsync** zabezpečuje spustenie exekúcie asynchrónnej operácie na pozadí, a teda exekúciu programových príkazov na pracovnom programovom vlákne:

```
backgroundWorker1.RunWorkerAsync();
```

Volanie metódy **RunWorkerAsync** spôsobí vznik udalosti **DoWork**, na ktorú bude aplikácia pomocou inštancie delegáta **DoWorkEventHandler** reagovať volaním metódy **backgroundWorker1_DoWork**. Po spustení asynchrónnej exekúcie na pracovnom programovom vlákne je možné z tela metódy **backgroundWorker1_DoWork** volať preťaženú inštančnú metódu **ReportProgress** triedy **BackgroundWorker**.

```
backgroundWorker1.ReportProgress(progresUlohy);
```

Táto metóda spôsobí vznik udalosti **ProgressChanged**, na ktorú reaguje inštancia delegáta **ProgressChangedEventHandler** volaním s ňou spojenej metódy **backgroundWorker1_ProgressChanged**. Preťažená metóda **ReportProgress** v našom prípade disponuje jedným formálnym parametrom typu **Int32**, ktorý predstavuje progres spracovania asynchrónnej operácie na pozadí, a teda progres spracovania na pracovnom programovom vlákne. Parameter môže nadobúdať hodnoty z intervalu <0, 100>.

Poslednou z vybraných metód je inštančná metóda **CancelAsync**, ktorej volanie spôsobí vznik požiadavky na ukončenie asynchrónnej operácie pomocou nastavenia skalárnej inštančnej vlastnosti **CancellationPending** triedy **BackgroundWorker** na hodnotu logickej pravdy **true**.

```
backgroundWorker1.CancelAsync();
```

Využitie metódy **CancelAsync** nie je nutnou podmienkou pre úspešnú implementáciu triedy **BackgroundWorker** v paralelnej aplikácii, napriek tomu ňou môžeme obohatiť funkcionalitu našej aplikácie.

1.2.4 Vybrané vlastnosti triedy BackgroundWorker

Prvou z často využívaných skalárnych inštančných vlastností je vlastnosť **IsBusy**, ktorá je určená iba na čítanie a môže uchovávať hodnotu logickej pravdy **true**, v prípade exekúcie asynchrónnej operácie a hodnotu logickej nepravdy **false** v prípade, že inštancia triedy **BackgroundWorker** nevykonáva žiadnu asynchrónnu operáciu.

Druhou frekventovane využívanou skalárnou inštančnou vlastnosťou je **CancellationPending**, ktorá je rovnako určená iba na čítanie. Vlastnosť uchováva hodnotu logickej pravdy **true** v prípade, že bola zavolaná metóda **CancelAsync**, a teda inštancia triedy **BackgroundWorker** prostredníctvom tejto metódy prijala požiadavku na ukončenie spracovania asynchrónnej operácie. V opačnom prípade je nastavená na logickú nepravdu **false**, ktorá je jej východiskovou hodnotou. Implementácia vlastnosti **CancellationPending** predpokladá overovanie jej hodnoty na pracovnom programovom vlákne a v prípade nastavenia vlastnosti na hodnotu logickej pravdy **true** zabezpečuje termináciu exekúcie na tomto programovom vlákne.

1.3 Knižnica Task Parallel Library

Knižnica **Task Parallel Library (TPL)** je významnou novinkou v oblasti paralelizácie algoritmov v prostredí vývojovo–exekučnej platformy Microsoft .NET Framework 4.0. Z pohľadu finálneho vývojára TPL predstavuje predovšetkým menný priestor **System.Threading.Tasks**, ktorého najvýznamnejšími súčasťami sú triedy **Task** a **Parallel**. Zatiaľ čo inštancie triedy **Task** sú reprezentáciami asynchrónnych úloh a implementujú tzv. **explicitný úlohový paralelizmus**, trieda **Parallel** je statickou triedou a jej statické metódy predstavujú paralelné implementácie dobre známych cyklov **for** a **foreach**, ktoré sú zamerané na **explicitný dátový paralelizmus**. Tretia statická metóda umožňuje simultánnu exekúciu viacerých programových činností, založenú na **explicitnom úlohovom paralelizme**. Nemenej významnou časťou TPL je menný priestor **System.Collections.Concurrent**, ktorý obsahuje špeciálne dátové štruktúry, inými slovami vláknovo bezpečné triedy kolekcií, ktoré by mali v prípade potreby v paralelných programoch nahradiť zodpovedajúce kolekcie z menných priestorov **System.Collections** a **System.Collections.Generic**.

Prostredníctvom knižnice TPL vývojár získava nielen ďalšie spôsoby ako paralelizovať exekúciu svojich sekvenčných aplikácií, alebo vytvárať od základov nové paralelné aplikácie, ale aj spôsob akým vykonávať všetky tieto úkony s využitím explicitného paralelizmu na vysokej úrovni abstrakcie. Konkurenčná výhoda, ktorú predstavuje vysoká produktivita práce, je dnes v softvérovom priemysle obzvlášť žiadaná a oceňovaná.

Použitie knižnice TPL uľahčuje dosiahnutie maximálneho využitia dostupného výpočtového výkonu hardvérovej platformy, na ktorej je paralelná aplikácia spúšťaná a navyše odbremeňuje vývojárov od

zabezpečovania realizácie všetkých technických služieb paralelnej exekúcie viacvláknovej riadenej aplikácie. Vývojár sa musí postarať iba o jediný predpoklad, ktorým je eliminácia závislostí medzi viacerými programovými vláknami. O zvyšné predpoklady súvisiace s optimálnym počtom programových vlákien, ich škálovateľnosťou a optimálnou distribúciou pracovného zaťaženia naprieč nimi sa postarajú pokročilé algoritmy plánovača úloh v spolupráci so spoločným behovým prostredím, ktoré knižnica TPL využíva.

1.4 Opis funkcionality inštancií triedy Task

Trieda **Task** z menného priestoru **System.Threading.Tasks** predstavuje objektovú abstrakciu úlohy. Použitie jej inštancií reprezentuje model explicitného paralelizmu založeného na úlohách – **Task Based Model**.

Úloha je reprezentovaná asynchrónnou operáciou, ktorá je podobná asynchrónnej operácii exekvovanej na programovom vlákne, úloha však operuje na vyššej úrovni abstrakcie. To znamená, že nie je priamo zviazaná s programovým vláknom. Plánovač úloh a spoločné behové prostredie zabezpečujú pre inštancie triedy **Task** vytvorenie optimálneho počtu programových vlákien, škálovateľnosť a optimálnu distribúciu pracovného zaťaženia, aby ich exekúcia bola na danej hardvérovej platforme čo možno najefektívnejšia. Pri tejto činnosti prihliadajú v záujme efektívnej exekúcie na aktuálne zaťaženie exekučných jadier procesora.

1.4.1 Inštanciácia triedy Task

```
Task uloha = new Task(Metoda);
```

Exekúcia tohto príkazu zabezpečí inicializáciu odkazovej premennej **uloha**, ktorej typom je trieda **Task**, odkazom na inštanciu tejto triedy. Signatúra použitého parametrického konštruktora triedy **Task** má tvar:

```
public Task(Action action)
```

Volaním tohto konštruktora pri inštanciácii triedy **Task** dochádza k implicitnej inštanciácii anonymného delegáta typu **Action**, ktorý je inicializovaný odkazom na cieľovú bezparametrickú metódu bez návratovej hodnoty s identifikátorom **Metoda**. Táto metóda bude neskôr podrobená exekúcii na pracovnom programovom vlákne.

Podobne, ako to bolo pri inštanciácii triedy **Thread**, je možné aj v prípade triedy **Task** prepísať jej inštanciačný príkaz bez zmeny funkcionality do tvaru, v ktorom dáme explicitne najavo inštanciáciu delegáta typu **Action** (prípadne pomocou λ–výrazu poskytneme parametrickému preťaženému inštančnému konštruktoru triedy **Task** odkaz na kompilátorom automaticky vytvorenú bezparametrickú anonymnú metódu). Pretože parametrický inštančný konštruktor triedy je preťažený, môžeme mu implicitne alebo explicitne poskytnúť odkaz na parametrickú cieľovú metódu. Jediným parametrom tejto metódy musí byť odkazová premenná typu **object**.

Okrem uvedených možností inštanciácie poskytuje trieda **Task** aj možnosť inštanciácie a priameho spustenia asynchrónnej exekúcie úlohy pomocou metódy **StartNew**.

```
Task uloha = Task.Factory.StartNew(Metoda);
```

Syntax príkazu je veľmi jednoduchá a programátorsky prívetivá, napriek tomu je potrebné bližšie vysvetliť jej sémantiku. Pomocou statickej vlastnosti **Factory** triedy **Task** získame odkaz na práve vytvorenú inštanciu triedy **TaskFactory**, na ktorej bodkovou notáciou voláme preťaženú inštančnú metódu **StartNew**. Jediným parametrom metódy **StartNew** je v našom prípade odkaz na implicitne vytvorenú inštanciu delegáta typu **Action**, ktorý v sebe zapuzdruje ďalší odkaz, a to odkaz na bezparametrickú metódu bez návratovej hodnoty s identifikátorom **Metoda**. Podobne, ako parametrický inštančný konštruktor triedy **Task**, aj metóda **StartNew** triedy **TaskFactory** je preťažená (to znamená, že jej parametrom môže byť aj cieľová metóda s jedným formálnym parametrom typu **object**).

Návratovou hodnotou metódy **StartNew** bude vo všetkých prípadoch odkaz na novo vytvorenú inštanciu triedy **Task**, ktorej exekúcia bola spustená ihneď po jej vytvorení.

1.4.2 Vybrané metódy triedy Task a ich použitie

Vzápätí po vytvorení inštancie triedy **Task** má vývojár paralelnej aplikácie k dispozícii úlohu, ktorá je pripravená na spustenie exekúcie s ňou asociovaných programových príkazov. Je nutné dodať, že toto neplatí v prípade použitia metódy **StartNew**, pretože po jej vykonaní je exekúcia úlohy už spustená. Spustenie exekúcie úlohy zabezpečíme volaním preťaženej inštančnej bezparametrickej metódy **Start** triedy **Task**.

```
uloha.Start();
```

Presnejšie povedané, tento programový príkaz nespustí priamo exekúciu úlohy, spustí iba plánovanie jej exekúcie a rozhodnutie o spustení exekúcie asynchrónnej úlohy zostáva na plánovači úloh a spoločnom behovom prostredí.

Čakanie na dokončenie exekúcie úlohy zabezpečíme pomocou preťaženej inštančnej metódy **Wait**.

```
uloha.Wait();
```

Zaujímavou obmenou metódy **Wait** je statická preťažená metóda **WaitAll**.

```
WaitAll(ulohy);
```

Parametrom metódy **WaitAll** je v tomto prípade odkaz na jednorozmerné pole inštancií triedy **Task**. Metóda zabezpečí čakanie na dokončenie exekúcie všetkých objektov v poli s identifikátorom **ulohy**.

1.4.3 Vybrané vlastnosti triedy Task

Trieda **Task** disponuje niekoľkými statickými a skalárnymi inštančnými vlastnosťami, ktoré umožňujú prístup k rozšírenej funkcionalite samotnej triedy a rovnako sú schopné monitorovať stavy vytvorených inštancií.

Skalárna inštančná vlastnosť **Id** určená iba na čítanie uchováva unikátny identifikátor úlohy, ktorého typom je integrálny systémový dátový typ **System.Int32**. Identifikátor je na požiadanie pridelený spoločným behovým prostredím a nemusí korešpondovať s poradím, v akom boli inštancie triedy **Task** vytvárané.

Pomocou statickej vlastnosti **Factory** určenej iba na čítanie získavame odkaz na inštanciu triedy **TaskFactory**, ktorá bola pri prístupe k vlastnosti **Factory** vytvorená pomocou bezparametrického inštančného konštruktora. Na tejto inštancii môžeme následne zavolať metódu **StartNew**.

Veľmi užitočnou vlastnosťou je skalárna inštančná vlastnosť **Status**, ktorá implementuje iba prístupovú metódu **get**, a je teda určená iba na čítanie. Vlastnosť uchováva hodnotu enumerátora typu **TaskStatus**, ktorý môže nadobúdať nasledujúce hodnoty:

- **Created** – úloha bola vytvorená, nebolo však spustené plánovanie jej exekúcie.
- **WaitingForActivation** – úloha čaká na svoju aktiváciu a spustenie plánovania exekúcie.
- **WaitingToRun** – plánovanie exekúcie úlohy bolo spustené, nie však jej samotná exekúcia.
- **Running** – prebieha proces exekúcie úlohy.
- **WaitingForChildrenToComplete** – úloha je v stave implicitného čakania na dokončenie exekúcie svojich podúloh.
- **RanToCompletion** – exekúcia úlohy bola úspešne ukončená.
- **Canceled** – úloha potvrdila zrušenie svojej exekúcie pomocou generovania výnimky **OperationCanceledException**.
- **Faulted** – ukončenie exekúcie úlohy spôsobila neošetrená výnimka.

Skalárna inštančná vlastnosť **IsCanceled** určená iba na čítanie uchováva hodnotu logickej pravdy **true** v prípade, že exekúcia úlohy bola ukončená z dôvodu jej zrušenia. V opačnom prípade je jej hodnotou logická nepravda **false**.

Ďalšou skalárnou inštančnou vlastnosťou určenou iba na čítanie je **IsCompleted**, ktorá uchováva hodnotu logickej pravdy **true** v prípade, že exekúcia asynchrónnej úlohy bola ukončená, a teda úloha sa nachádza v jednom z finálnych stavov **RanToCompletion**, **Faulted** alebo **Canceled**. V opačnom prípade je jej hodnotou logická nepravda **false**.

Exception predstavuje ďalšiu z množiny skalárnych inštančných vlastností triedy **Task**. Takisto ako ostatné, aj ona je určená iba na čítanie a jej obsahom môže byť inštancia triedy **AggregateException**, ktorá predstavuje chybovú výnimku, ktorá vznikla počas exekúcie a ktorá spôsobí predčasné ukončenie exekúcie úlohy. Ak bola exekúcia úlohy úspešne ukončená, vlastnosť bude obsahovať nulovú referenciu **null**.

Poslednou zo skalárnych inštančných vlastností je **IsFaulted**. Je určená iba na čítanie a je nastavená na logickú pravdu **true** v prípade, že stav úlohy je nastavený na **Faulted** a vlastnosť **Exception** neobsahuje **null**. Inak je jej hodnotou **false**.

1.5 Opis funkcionality triedy Parallel

Statická trieda **Parallel** situovaná v mennom priestore **System.Threading.Tasks** zapuzdruje tri preťažené statické metódy, ktoré umožňujú dosiahnuť pri vývoji paralelných aplikácií vysokú úroveň abstrakcie. To znamená, že podobne ako pri práci s inštanciami triedy **Task** zabezpečuje plánovač úloh v spolupráci so spoločným behovým prostredím mnoho nízkoúrovňových softvérových služieb.

1.5.1 Vybrané metódy triedy Parallel a ich použitie

Preťažená statická metóda **For** predstavuje paralelnú implementáciu cyklu **for** jazyka C# 4.0. Túto metódu je vhodné použiť v prípade, že v aplikácii potrebujeme paralelne vykonať isté identické operácie nad prvkami poľa prípadne kolekcie. Samozrejme, je potrebné z algoritmu vylúčiť akékoľvek zdieľané zdroje a závislosť výpočtu medzi jednotlivými iteráciami cyklu, alebo využiť široké možnosti explicitnej synchronizácie, ktoré nám ponúka platforma .NET Framework 4.0 a jazyk C# 4.0. V druhom prípade to však bude na úkor celkového výkonu aplikácie, ktorý zníži explicitná synchronizácia.

```
Parallel.For(0, hornaHranica, i =>
{
        poleObjektov[i].Metoda();
});
```

Statická metóda **For** triedy **Parallel** očakáva nasledujúce argumenty:

- **fromInclusive** – je prvým argumentom (v našom prípade s hodnotou 0) typu **System.Int32** a predstavuje spodnú inkluzívnu hodnotu iteračnej premennej **i**.
- **toExclusive** – premenná s identifikátorom **hornaHranica** typu **System.Int32**, ktorá bude hornou exkluzívnou hodnotou iteračnej premennej **i**.
- **body** – je inštanciou delegáta typu **Action**, ktorý bude uchovávať odkaz na parametrickú cieľovú metódu bez návratovej hodnoty. Jediný parameter tejto metódy bude typu **System.Int32**. Na tomto mieste sa použije λ–výraz, pomocou ktorého je kompilátorom automaticky vytvorená anonymná metóda podľa uvedených požiadaviek.

Hodnota premennej **i** bude v každej iterácii paralelného cyklu **for**[2] odovzdávaná v podobe argumentu prekladačom vytvorenej anonymnej metóde. Výsledkom bude volanie inštančnej metódy s identifikátorom **Metoda** paralelne nad zvoleným počtom objektov poli s identifikátorom **poleObjektov**.

[2] Paralelný cyklus **for** je zaužívané pomenovanie pre statickú metódu **For** triedy **Parallel**.

Druhou preťaženou statickou metódou triedy **Parallel** je metóda **ForEach**, ktorá je paralelnou implementáciou cyklu **foreach** jazyka C# 4.0. Účel použitia a princípy paralelizácie exekúcie metódy **ForEach** sú analogické s metódou **For**, avšak metóda **ForEach** automaticky zabezpečí iteráciu naprieč všetkými prvkami poľa, respektíve kolekcie. Rovnako, ako v prípade použitia cyklu **foreach** jazyka C# 4.0, je nutná implementácia rozhrania **IEnumerable** z menného priestoru **System.Collections.Generic** uvažovaným poľom alebo kolekciou objektov.

```
Parallel.ForEach(kolekciaObjektov, objekt =>
{
       objekt.Metoda();
});
```

Argumenty, s ktorými je metóda **ForEach** volaná, sú takéto:

- **source –** predstavuje pole alebo kolekciu objektov, ktorá implementuje rozhranie **IEnumerable**.

- **body** – je inštanciou delegáta typu **Action**, ktorý bude uchovávať odkaz na parametrickú cieľovú metódu bez návratovej hodnoty. Parametrom tejto metódy však bude inštancia bázovej triedy **Object** z menného priestoru **System**.

Výsledkom uvedeného paralelného cyklu **ForEach** bude volanie inštančnej metódy s identifikátorom **Metoda** paralelne, nad každým objektom kolekcie s identifikátorom **kolekciaObjektov**.

Poslednou z trojice statických metód triedy **Parallel** je preťažená metóda **Invoke**. Kým v prípade dvoch predchádzajúcich metód sme hovorili o explicitnom dátovom paralelizme, metóda **Invoke** implementuje explicitný úlohový model paralelizmu.

```
Parallel.Invoke(Metoda1, Metoda2, Metoda3);
```

Jediným argumentom preťaženej metódy **Invoke** je jednorozmerné pole troch automaticky prekladačom vytvorených inštancií delegátov typu **Action**. Každý z nich zapuzdruje vo svojom tele odkaz na jednu z bezparametrických cieľových metód bez návratových hodnôt s identifikátormi **Metoda1**, **Metoda2**, **Metoda3**.

Pre takto vytvorené úlohy je po volaní metódy **Invoke** spustené plánovanie ich exekúcie, pričom počet programových vlákien, škálovateľnosť a distribúcia záťaže je v kompetencii plánovača úloh a spoločného behového prostredia. V prípade, že cieľové metódy pracujú so zdieľanými zdrojmi, je nutné zabezpečiť ich explicitnú synchronizáciu. Exekúcia na programovom vlákne, z ktorého bola metóda **Invoke** zavolaná, bude pozastavená až do okamihu ukončenia exekúcie všetkých požadovaných úloh. Napriek všetkému, neexistuje zo strany plánovača úloh a spoločného behového prostredia žiadna záruka, že uvedené metódy budú skutočne vykonané paralelne.

1.6 Špecifické dátové štruktúry a synchronizačné primitíva

Z pohľadu využitia paradigmy paralelného objektového programovania na vývojovo–exekučnej platforme .NET Framework 4.0 môžeme za špecifické označiť triedy kolekcií z menného priestoru **System.Collections.Concurrent**. Tieto triedy kolekcií umožňujú bezpečný prístup k svojim prvkom z viacerých vlákien, čo následne odbúrava nutnosť explicitnej synchronizácie vlákien pri manipulačných prácach s týmito kolekciami. Táto vlastnosť ich predurčuje na použitie v paralelných aplikáciách.

Príkladom triedy, ktorá implementuje vláknovo bezpečnú kolekciu, je trieda **ConcurrentDictionary**, ktorá si osvojila princíp slovníka ako dátovej štruktúry. Inštancie tejto kolekcie predstavujú súbor párov kľúč – hodnota, ku ktorým je možné bezpečne pristupovať v jednom časovom okamihu z viacerých programových vlákien. Trieda zároveň implementuje rozhrania ako **IDictionary**, **ICollection** a **IEnumerable**, čo ju zaväzuje definovať všetky členy, ktoré sú v uvedených rozhraniach deklarované.

Medzi ďalšie vláknovo bezpečné triedy kolekcií patria napríklad:

- **BlockingCollection(T[3])** – umožňuje vláknovo bezpečné pridávanie a odoberanie prvkov.
- **ConcurrentBag(T)** – reprezentuje neusporiadanú vláknovo bezpečnú kolekciu objektov.
- **ConcurrentQueue(T)** – predstavuje kolekciu typu prvý dnu – prvý von (first in – first out, FIFO), je príkladom implementácie radu ako dátovej štruktúry.
- **ConcurrentStack(T)** – je kolekciou typu posledný dnu – prvý von (last in – first out, LIFO), čiže zásobník.

Ak povaha implementovaného problému vylučuje použitie tried kolekcií z menného priestoru **System.Collections.Concurrent**, je možné použiť niektoré z dostupných synchronizačných primitív, ako napríklad atomické operácie, monitory, mutexy a rôzne synchronizačné udalosti.

Často využívané atomické operácie predstavujú spracovanie množiny čiastkových operácií v rámci jednej diskrétnej logickej jednotky, bez možnosti prerušenia exekúcie. To znamená, že na všetky čiastkové operácie môžeme v danom programovom príkaze nahliadať ako na jedinú a nedeliteľnú operáciu[4]. Spôsobom, akým je možné atomickú operáciu implementovať, je použitie statickej triedy **Interlocked** z menného priestoru **System.Threading**.

Trieda **Interlocked** umožňuje pomocou svojich metód vykonanie nasledujúceho súboru operácií:

- Atomické sčítanie hodnôt premenných systémových dátových typov **System.Int32** a **System.Int64**.
- Atomickú inkrementáciu a dekrementáciu hodnôt premenných systémových dátových typov **System.Int32** a **System.Int64**.
- Atomickú operáciu priradenia hodnoty do premenných typu **System.Double**, **System.Int32**, **System.Int64**, **System.IntPtr**, **System.Object** a **System.Single**, prípadne akéhokoľvek odkazového dátového typu.

[3] **T** predstavuje názov odkazového dátového typu prvku kolekcie.
[4] V princípe sú atomické operácie najviac podobné transakčnému spracovaniu v oblasti relačných databáz.

- Atomickú operáciu porovnania hodnôt dvoch premenných typu **System.Double**, **System.Int32**, **System.Int64**, **System.IntPtr**, **System.Object** a **System.Single**, prípadne akéhokoľvek iného odkazového dátového typu a v prípade zhody priradenie špecifikovanej hodnoty do prvej z nich.
- Atomickú operáciu čítania hodnoty premennej typu **System.Int64**[5].

2 Vývoj paralelnej aplikácie v jazyku C# 4.0

Paralelná aplikácia, ktorá je predmetom tohto praktického cvičenia, sa zaoberá odhaľovaním šifrovaných prístupových hesiel do informačných systémov chránených technikou hašovania. Aplikácia využíva viacero spôsobov paralelizácie svojej činnosti, čím poukazuje na rôzne aspekty ich využitia a s nimi súvisiacu dosahovanú efektivitu.

2.1 Analytické zhrnutie riešenia

Jednou z ciest narušenia bezpečnosti informačného systému je útok na informačný systém vedený z vnútra organizácie prevádzkujúcej informačný systém. Tento typ útoku môže byť okrem iného zameraný na neoprávnené získanie prístupu k používateľským kontám. Charakteristickým znakom tohto typu útoku je predovšetkým veľmi dobrá znalosť informačného systému zo strany útočníka, a teda osoby, ktorá bezpečnosť informačného systému ohrozuje.

Dnešné informačné systémy vo svojich databázach spravidla uchovávajú autentifikačné údaje o svojich používateľoch v podobe používateľského mena a sekvencie textových znakov reprezentujúcich výstup istej hašovacej funkcie po jej aplikácii, napríklad na kombináciu používateľského mena a hesla.

Hašovacia funkcia vo všeobecnosti predstavuje matematickú funkciu, na ktorej vstupe sa môže nachádzať ľubovoľný objem dát a ktorej výstupom budú vždy dáta konštantného objemu. Jednou z kľúčových vlastností hašovacích funkcií je veľmi veľká výpočtová náročnosť, v prípade ideálnej hašovacej funkcie nemožnosť odvodenia vstupných dát z jej výstupu. Zároveň musí byť výstup hašovacej funkcie[6] deterministický a nesmie nastať jeho kolízia pri rôznych vstupoch hašovacej funkcie. Tieto vlastnosti hašovacích funkcií zabezpečili ich široké uplatnenie v oblasti bezpečnosti informačných systémov.

Vo všeobecnosti teda proces autentifikácie používateľa do informačného systému prebieha v nasledujúcich krokoch:

- Vloženie používateľského mena a hesla do príslušného autentifikačného formulára.
- Prevedenie kombinácie používateľského mena a hesla na sekvenciu textových znakov konštantnej dĺžky pomocou zvolenej hašovacej funkcie t.j. výpočet hašu.

[5] Atomická operácia čítania hodnoty premennej systémového typu **System.Int64** má zmysel iba v 32–bitových operačných systémoch, v ktorých je implicitne zabezpečená atomicita čítania hodnôt premenných typu **System.Int32**.
[6] Pre výstup hašovacej funkcie je zaužívaný názov **haš**.

- Porovnanie používateľom zadaného používateľského mena a hašu s používateľským menom a hašom uloženým v samotnej databáze informačného systému.
- Povolenie, prípadne zamietnutie prístupu do informačného systému na základe porovnania uskutočneného v predchádzajúcom kroku.

Ak je teda naším cieľom vytvorenie aplikácie, ktorá bude schopná odhaľovať prístupové heslá používateľov do informačných systémov, musíme prijať nasledujúce predpoklady o znalostiach útočníka:

- Útočníkovi je známe používateľské meno a haš ako výstup hašovacej funkcie po jej aplikácii napríklad na kombináciu používateľského mena a hesla.
- Má informácie o type použitej hašovacej funkcie a spôsobe jej použitia.
- Ďalšími informáciami, ktorými disponuje, sú typ a dĺžka použitého hesla.

2.2 Návrh riešenia

Pre potrebu efektívnej správy vstupných dát našej aplikácie sme si zvolili pre vstupné údaje súborový formát Microsoft Excel. Prístup k vstupným súborom zabezpečí rozhranie **OLE DB**, ktoré predstavuje unifikovaný štandard pre manipuláciu s rôznymi dátovými zdrojmi. Zároveň sa vyznačuje jednoduchosťou použitia a rýchlosťou prístupu k údajom. Toto rozhranie umožní aplikácii prácu so súborovým formátom **xls,** ako aj s novým formátom **xlsx** používaným od verzie Microsoft Excel 2007. Použitie rozhrania **OLE DB** implikuje použitie tried zapuzdrených v menných priestoroch **System.Data** a **System.Data.OleDb**.

V aplikácii sme sa rozhodli implementovať hašovacie funkcie **MD5** a **SHA-2**. Funkcia **MD5** bola napriek jej známym nedostatkom vybraná z dôvodu stále veľkého rozšírenia a nenáročnosti na výpočet, čo ju predurčuje na demonštráciu v aplikácii. Naopak funkciu **SHA-2** vo verzii **SHA256** sme vybrali v súčasnosti ako odporúčané riešenie na poli kryptografických hašovacích funkcií a z toho plynúci potenciál stať sa rozšírenou v priebehu niekoľkých rokov. Aplikáciu samozrejme navrhneme tak, aby bolo v budúcnosti možné jej rozšírenie o akúkoľvek hašovaciu funkciu podporovanú vývojovo–exekučnou platformou .NET Framework 4.0.

Požiadavku na schopnosť práce s hašovacími funkciami **MD5** a **SHA-2** zabezpečí import menného priestoru **System.Security.Cryptography**, ktorý združuje triedy umožňujúce implementáciu potrebnej funkcionality.

Hlavným cieľom navrhovanej aplikácie, ktorým je porovnanie sekvenčného a paralelných algoritmov hľadania hesiel, dosiahneme implementáciou jedného sekvenčného a štyroch typov paralelných algoritmov. Pomocou tohto riešenia budeme schopní priamo demonštrovať rôznu úroveň abstrakcie jednotlivých algoritmov a súvisiacu náročnosť a efektivitu vývoja týchto algoritmov.

Medzi uvedené štyri typy paralelných algoritmov patria:

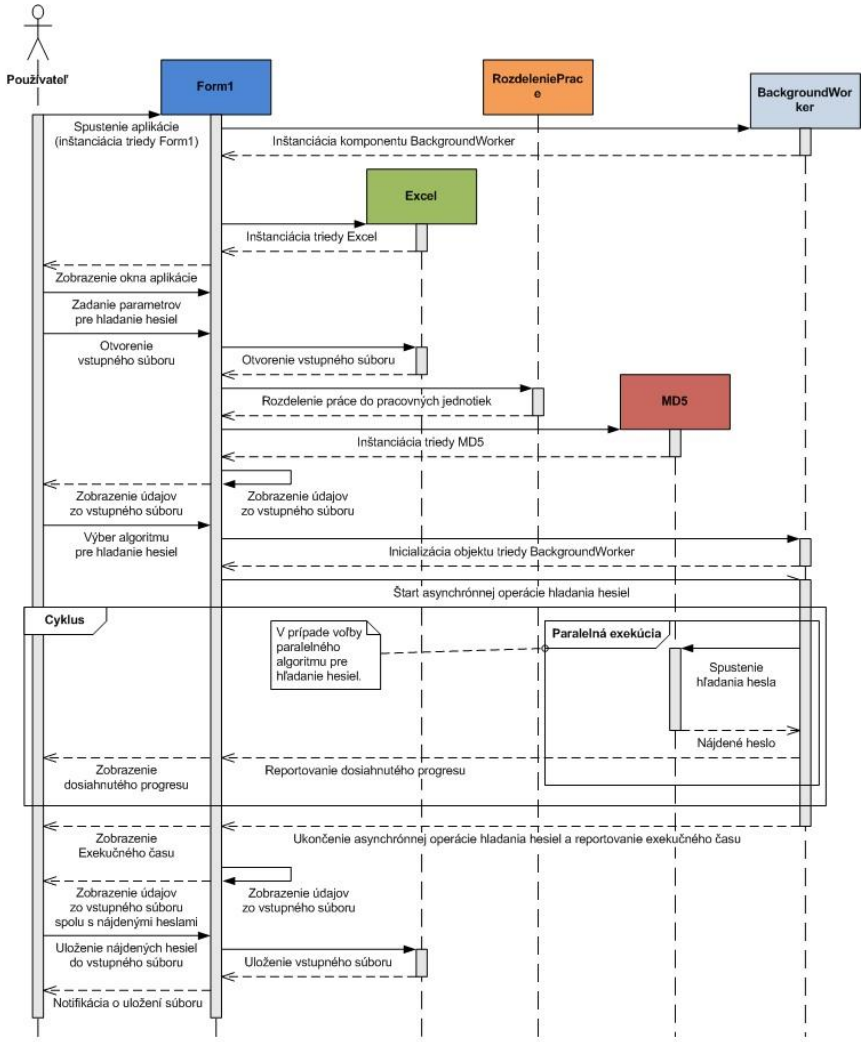
- Algoritmus využívajúci inštancie triedy **Thread** – explicitný paralelizmus s nízkou úrovňou abstrakcie.
- Algoritmus využívajúci inštancie triedy **Task** – explicitný úlohový paralelizmus so strednou úrovňou abstrakcie.
- Algoritmus využívajúci pri paralelizácii svojich činností paralelný cyklus **For** – explicitný dátový paralelizmus s vysokou úrovňou abstrakcie.
- Algoritmus využívajúci pri paralelizácii svojich činností paralelný cyklus **ForEach** – explicitný dátový paralelizmus s vysokou úrovňou abstrakcie.

2.2.1 Diagram tried paralelnej aplikácie

Obr. 1: Diagram tried paralelnej aplikácie **Hľadanie hesiel**

Parciálna trieda **Form1** je triedou formulára hlavného aplikačného okna. Dátovými položkami tejto triedy sú okrem všetkých vizuálnych prvkov formulára tiež inštancie tried **Excel**, **MD5**, **SHA_2** a inštancia komponentu **BackgroundWorker**. Úlohou triedy **Excel** je zabezpečiť prístup k vstupnému súboru pomocou rozhrania **OLE DB**, triedy **MD5** a **SHA_2** umožnia nájdenie hesla pre zadanú kombináciu používateľského mena (alebo iného doplnkového textového reťazca[7]) a používateľovi známeho výstupu hašovacej funkcie v podobe hašu, z ktorého chce heslo odvodiť. Diagram tried zobrazuje okrem uvedených tried ešte statickú triedu **RozdeleniePrace**, ktorá slúži na rozdelenie súboru šifrovaných hesiel do takého počtu pracovných jednotiek, aby bolo v prípade paralelného algoritmu využívajúceho inštancie triedy **Thread** zabezpečené vytvorenie optimálneho počtu pracovných programových vlákien a optimálna distribúcia pracovného zaťaženia naprieč všetkými vytvorenými vláknami, čo je potrebné z dôvodu neskoršej reprezentácie každej vytvorenej pracovnej jednotky pracovným programovým vláknom.

2.2.2 Sekvenčný diagram paralelnej aplikácie



Obr. 2: Sekvenčný diagram paralelnej aplikácie **Hľadanie hesiel**

[7] V anglickej literatúre je tento doplnkový text, ktorý výrazne zvyšuje bezpečnosť hesiel uložených v databázach informačných systémov, označovaný všeobecne akceptovaným termínom **salt**.

Sekvenčný diagram prehľadne definuje kľúčové interakcie medzi objektmi tried **Form1**, **Excel**, **MD5**, **BackgroundWorker** a statickou triedou **RozdeleniePrace** potrebné na pochopenie návrhu aplikácie. Sekvenčný diagram zobrazuje interakcie s triedou **MD5**, pričom v prípade použitia triedy **SHA_2**, ktoré sa v aplikácii vyskytuje, by bol diagram identický s tým, že triedu **MD5** by v ňom nahradila trieda **SHA_2**. Ďalším špecifikom sekvenčného diagramu na obr. 2 je zobrazenie iba jednej iterácie procesu hľadania hesiel. To znamená, že zobrazujeme proces hľadania hesiel pre jeden vstupný súbor. Samozrejmosťou je, že používateľ môže tento proces uskutočniť pre ľubovoľný počet vstupných súborov bez nutnosti opätovného spustenia aplikácie.

2.2.3 Návrh testov paralelnej aplikácie

Pre väčšiu názornosť aplikáciu otestujeme na dvoch počítačoch s rôznymi viacjadrovými procesormi v nasledujúcich konfiguráciách:

- Prenosný počítač osadený 2-jadrovým procesorom Intel Core 2 Duo P7370. Procesor je vyrábaný 45-nanometrovou technológiou, disponuje vyrovnávacou pamäťou L1-cache o veľkosti 128 KB, L2-cache o veľkosti 3 MB a každé z jeho exekučných jadier pracuje na frekvencii 2,0 GHz. Konfiguráciu prenosného počítača dopĺňajú 4 GB operačnej pamäte RAM a 32-bitový operačný systém Microsoft Windows 7.

- Stolový počítač osadený 4-jadrovým procesorom Intel Core i5 750. Procesor je vyrábaný 45-nanometrovou technológiou, disponuje vyrovnávacou pamäťou L1-cache o veľkosti 256 KB, L2-cache o veľkosti 1 MB a L3-cache spoločnej pre všetky exekučné jadrá s veľkosťou 8 MB. Každé z exekučných jadier vo východiskovom stave pracuje na frekvencii 2,66 GHz, v prípade potreby však procesor dokáže automaticky zvýšiť svoju pracovnú frekvenciu na 3,2 GHz. Konfiguráciu dopĺňajú 4 GB operačnej pamäte RAM a 64-bitový operačný systém Microsoft Windows 7.

Prvá časť testov sa zameria na diagnostiku paralelnej aplikácie pomocou nástrojov integrovaných vo vývojovom prostredí Visual Studio 2010. Ide predovšetkým o profilovací nástroj, ktorý nám poskytne výstup v podobe výkazu s názvom **Concurrency Profiling Report**. Prostredníctvom vygenerovaného výkazu budeme schopní analyzovať tri charakteristiky našej paralelnej aplikácie:

- Schopnosť paralelnej aplikácie využiť dostupné exekučné jadrá viacjadrového procesora – okno **CPU Utilization**.
- Stavy jednotlivých programových vlákien počas životného cyklu paralelnej aplikácie – okno **Threads**.
- Migráciu programových vlákien[8] medzi exekučnými jadrami viacjadrového procesora – okno **Cores**.

Na diagnostiku paralelnej aplikácie pomocou profilovacieho nástroja vývojového prostredia Visual Studio 2010 použijeme prvú konfiguráciu počítača s dvomi exekučnými jadrami procesora a vstupný súbor 6 používateľských mien a výstupov hašovacej funkcie **MD5** po jej aplikácii na kombináciu

[8] V literatúre sa pomerne často môžeme stretnúť aj s termínom **afinita programových vlákien**.

používateľského mena a hesla. Diagnostiku paralelnej aplikácie prevedieme s algoritmom využívajúcim na paralelizáciu svojich činností paralelný cyklus **for**.

Jednou z funkcionalít paralelnej aplikácie **Hľadanie hesiel** bude reportovanie dĺžky exekučných časov jednotlivých algoritmov použitých na nájdenie súboru šifrovaných hesiel, o ktoré sa bude opierať druhá časť testov. Na základe týchto údajov môžeme teda uskutočniť porovnávací test všetkých druhov implementovaných algoritmov. Každý test algoritmu sa bude skladať z piatich meraní, výsledný exekučný čas algoritmu bude vypočítaný aritmetickým priemerom z piatich nameraných hodnôt. Súpravu testov uskutočníme na oboch konfiguráciách počítačov so vstupnými súbormi 6 používateľských mien a výstupov hašovacej funkcie. Otestujeme hašovaciu funkciu **MD5** aj **SHA-2**.

Z nameraných výkonnostných charakteristík algoritmov vypočítame **nárast výkonnosti paralelného algoritmu** v porovnaní s algoritmom implementujúcim sekvenčné spracovanie programových príkazov.

$$N_V = \frac{T_S}{T_P}$$

kde:

- N_V je nárast výkonnosti paralelného algoritmu v porovnaní so sekvenčným algoritmom,
- T_S je priemerným exekučným časom sekvenčného algoritmu,
- T_P je priemerným exekučným časom paralelného algoritmu.

Hodnoty nárastu výkonnosti paralelného programu vieme klasifikovať ako:

- **Sublineárny** nárast výkonnosti, ak $N_V < n$.
- **Lineárny** nárast výkonnosti, ak $N_V = n$.
- **Superlineárny** nárast výkonnosti, ak $N_V > n$.

kde:

- n je počet exekučných jadier procesora, na ktorom je paralelná aplikácia exekvovaná.

Vzťah na **výpočet efektivity využitia výpočtových zdrojov** počítačového systému, umožňuje porovnať efektivitu sekvenčných a paralelných algoritmov:

$$e = \left(\frac{N_V}{n}\right) \times 100\ [\%]$$

kde:

- e je efektivita využitia výpočtových zdrojov počítačového systému v percentách,
- N_V je nárast výkonnosti paralelnej aplikácie v porovnaní so sekvenčnou aplikáciou,
- n predstavuje počet exekučných jadier viacjadrového procesora.

Efektivita využitia výpočtových zdrojov počítačového systému vyjadruje priemernú mieru využívania exekučných jadier procesora počas exekúcie istého algoritmu[9].

2.2.4 Softvérové požiadavky na cieľové pracovné stanice

Na základe navrhnutej špecifikácie paralelnej aplikácie **Hľadanie hesiel** uvádzame softvérové požiadavky pre beh aplikácie na cieľových pracovných staniciach:

- Operačný systém Microsoft Windows XP, Windows Vista, Windows 7, Windows Server 2003 alebo Windows Server 2008.
- Vývojovo–exekučná platforma Microsoft .NET Framework 4.0 alebo .NET Framework 4.0 Client Profile.
- Kancelársky balík Microsoft Office 2007, alebo súbor komponentov Office 2007 System Driver, ktoré umožňujú prácu so súbormi aplikácií Office 2007.

2.3 Implementácia riešenia

2.3.1 Trieda Excel

Prvou z tried, ktorú v aplikácii implementujeme, je trieda **Excel**, ktorej úlohou je zabezpečovať aplikácii prístup k súboru vo formáte Microsoft Excel pomocou rozhrania **OLE DB**. Trieda disponuje niekoľkými súkromnými dátovými položkami potrebnými na úspešnú implementáciu rozhrania. Jediná skalárna inštančná vlastnosť triedy je vlastnosť s identifikátorom **Tabuľka**, ktorá umožňuje klientskemu programovému kódu prístup k rovnomennej súkromnej dátovej položke, ktorej typom je trieda **DataTable**. Pomocou tejto dvojice získava vývojár prístup ku kolekcii riadkov, ktoré budú naplnené údajmi zo vstupného súboru aplikácie.

Metóda OtvoritSubor

```
public void OtvoritSubor(string nazovSuboru)
{
	StringBuilder pripojovaciRetazec = new StringBuilder();
	string chybovaSprava = "Súbor programu Excel nemá správnu štruktúru!";
	datovySubor.Clear();
	pripojovaciRetazec.Append(@"Provider=Microsoft.ACE.OLEDB.12.0;Data
	Source=");
	pripojovaciRetazec.Append(nazovSuboru);
	pripojovaciRetazec.Append(@";Extended Properties=""Excel 12.0;HDR=NO;""");
	try
	{
		spojenie.ConnectionString = pripojovaciRetazec.ToString();
		if (spojenie.State == ConnectionState.Closed)
			spojenie.Open();

		prikaz.Connection = spojenie;
		adapter.SelectCommand = prikaz;
```

[9] HANÁK, J. 2009. *Praktické paralelné programovanie v jazykoch C# 4.0 a C++*. Brno: Artax, 2009. 132 s. ISBN 978-80-87017-06-7.

```
                VykonajSqlPrikaz("SELECT * FROM [Sheet1$A1:C1]");
                if (tabulka.Rows[0][0].ToString() != "Meno")
                        throw new Exception(chybovaSprava);
                if (tabulka.Rows[0][1].ToString() != "Hash")
                        throw new Exception(chybovaSprava);
                if (tabulka.Rows[0][2].ToString() != "Heslo")
                        throw new Exception(chybovaSprava);

                VykonajSqlPrikaz("SELECT * FROM [Sheet1$A2:B65536]");
        }
        catch (Exception)
        {
                UkonciSpojenie();
                throw;
        }
}
```

Jediným formálnym parametrom metódy je názov súboru s identifikátorom **nazovSuboru** typu **String**, ktorého otvorenie je klientskym programovým kódom požadované. V tele metódy je pomocou inštancie triedy **StringBuilder** najskôr skonštruovaný reťazec znakov, aby vo vetve **try** programovej konštrukcie **try–catch** bola naplnená skalárna inštančná vlastnosť **ConnectionString** dátovej položky **spojenie**, ktorej typom je trieda **OleDbConnection**. Ďalej nasleduje správa chybových výnimiek pre prípad nesprávnej štruktúry vstupného súboru. V prípade, že je štruktúra vstupného súboru v poriadku, vykoná sa príkaz **SELECT** na vstupné údaje pre aplikáciu. Ak vo vetve **try** nastane akákoľvek chybová výnimka, je spojenie so vstupným súborom ukončené a vzniknutá chybová výnimka je propagovaná klientskemu programovému kódu.

Metóda UlozitSubor

Metóda s identifikátorom **UlozitSubor** zabezpečí v tele cyklu **for** vykonanie príkazu **UPDATE** za účelom aktualizácie vstupného súboru aplikácie o zoznam nájdených používateľských hesiel, ktorý metóda získa v podobe svojho formálneho parametra, a teda odkazu na inštanciu generickej kolekcie typu **List<string>**. Po aktualizácii záznamov metóda ukončí spojenie so vstupným súborom.

```
public void UlozitSubor(List<string> zoznam)
{
        try
        {
                for (int i = 0; i < zoznam.Count; i++)
                        VykonajSqlPrikaz("UPDATE [Sheet1$C" + (i + 2).ToString() +
                        ":C" + (i + 2).ToString() + "] SET F1 = '" + zoznam[i]
                        + "'");

                        UkonciSpojenie();
        }
        catch (Exception)
        {
                UkonciSpojenie();
                throw;
        }
}
```

2.3.2 Trieda RozdeleniePrace

Úlohou statickej triedy **RozdeleniePrace** s jedinou statickou metódou **RozdelPracu** je, v prípade použitia algoritmu využívajúceho inštancie triedy **Thread** vytvoriť zodpovedajúci počet pracovných jednotiek a dohliadnuť na rozdelenie celkového množstva práce medzi pracovné jednotky tak, aby ich zaťaženie bolo rovnomerné.

```
public static void RozdelPracu(int mnozstvoPraceNaRozdelenie)
{
	celkoveMnozstvoPrace = mnozstvoPraceNaRozdelenie;
	if (celkoveMnozstvoPrace > Environment.ProcessorCount)
	{
		pocetPracovnychJednotiek = Environment.ProcessorCount;
		mnozstvoPraceVPracovnejJednotke = celkoveMnozstvoPrace /
			pocetPracovnychJednotiek;
	}
	else
	{
		pocetPracovnychJednotiek = celkoveMnozstvoPrace;
		mnozstvoPraceVPracovnejJednotke = 1;
	}
}
```

Formálnym parametrom metódy je celkové množstvo práce, ktoré má byť medzi pracovné jednotky rozdelené, pričom môžu nastať dva stavy:

- Celkové množstvo práce je väčšie ako počet hardvérových vlákien podporovaných danou platformou. Túto hodnotu reprezentuje vlastnosť **ProcessorCount** statickej triedy **Environment** z menného priestoru **System**. V tomto prípade je vytvorený počet pracovných jednotiek rovný počtu podporovaných hardvérových vlákien. Množstvo práce v každej vytvorenej pracovnej jednotke s výnimkou v poradí poslednej pracovnej jednotky, je dané výsledkom celočíselného delenia celkového množstva práce a počtu vytvorených pracovných jednotiek. Množstvo práce v poslednej pracovnej jednotke bude pri vytváraní pracovných jednotiek určovať rozdiel celkového množstva práce a množstva práce, ktoré bolo pridelené spolu všetkým ostatným pracovným jednotkám.

- Celkové množstvo práce je menšie, alebo rovné počtu podporovaných hardvérových vlákien. Vytvorený počet pracovných jednotiek sa v tomto prípade bude rovnať celkovému množstvu práce. Množstvo práce v každej, aj v poslednej pracovnej jednotke, sa bude rovnať jednej.

2.3.3 Triedy MD5 a SHA_2

Implementácia triedy **MD5** v paralelnej aplikácii **Hľadanie hesiel** je zameraná na nájdenie hesla za predpokladu poskytnutia používateľského mena a hašu, ktorý bol vypočítaný z kombinácie používateľského mena a hesla.

Metóda Zasifruj

Jednou z dátových položiek triedy je inštancia triedy **MD5CryptoServiceProvider**, ktorá zabezpečuje samotné hašovanie a jadro svojej činnosti v rámci našej aplikácie vykonáva práve v metóde **Zasifruj**:

```
private string Zasifruj(string meno, string heslo)
{
	bajtovePole = koder.GetBytes(heslo + meno);
	bajtovePole = md5CryptoServiceProvider.ComputeHash(bajtovePole);
	return Convert.ToBase64String(bajtovePole);
}
```

Metóda aplikuje hašovaciu funkciu **MD5** na používateľské meno a heslo, ktoré získa v podobe svojich formálnych parametrov. Návratovou hodnotou metódy je vypočítaný haš reprezentovaný textovým reťazcom.

Metóda NajdiHeslo

Z hľadiska pochopenia funkcionality triedy je kľúčovým nasledujúci fragment zdrojového kódu, ktorý je súčasťou inštančnej verejne prístupnej parametrickej metódy bez návratovej hodnoty s identifikátorom **NajdiHeslo**:

```
for (int i = 0; i < 1000000; i++)
{
	kandidatNaHeslo = i.ToString("D6");
	if (Zasifruj(this.meno, kandidatNaHeslo) == this.hash)
	{
		this.heslo = kandidatNaHeslo;
		break;
	}
}
```

Fragment zdrojového kódu predstavuje cyklus **for** s jedným miliónom iterácií. Ak si uvedomíme, že iteračná premenná **i** bude nadobúdať hodnoty z intervalu <0, 999999>, môžeme z nej pomocou vhodného formátovania na textový reťazec urobiť kandidáta na šesťmiestne heslo pozostávajúce výlučne z číslic. Ako vhodné formátovanie sa v našom prípade javí doplnenie hodnoty iteračnej premennej zľava nulami tak, aby bol vždy vytvorený textový reťazec s dĺžkou šesť znakov. Ďalším krokom je zašifrovanie obsahu dátovej položky **meno** spolu s kandidátom na heslo v aktuálnej iterácii cyklu a porovnanie návratovej hodnoty metódy **Zasifruj** s obsahom dátovej položky **hash**. Ak nastane zhoda, úspešného kandidáta na heslo uchováme v dátovej položke **heslo**. Hľadanie hesla pre aktuálnu inštanciu triedy **MD5** je týmto krokom ukončené.

V prípade triedy **SHA_2** je jej implementácia totožná s triedou **MD5**, jediným rozdielom medzi týmito dvoma triedami je použitie triedy **SHA256CryptoServiceProvider** na zabezpečenie samotného hašovania v metóde **Zasifruj**.

2.3.4 Trieda Form1

Najrozsiahlejšou triedou v paralelnej aplikácii **Hľadanie hesiel** je samotná parciálna trieda formulára hlavného aplikačného okna **Form1**. Jej primárnou funkciou je reagovať na rôzne udalosti, ktoré vznikajú počas celého životného cyklu aplikácie. Ďalšou dôležitou funkciou, ktorú vykonáva, je meranie exekučných časov jednotlivých algoritmov hľadania hesiel.

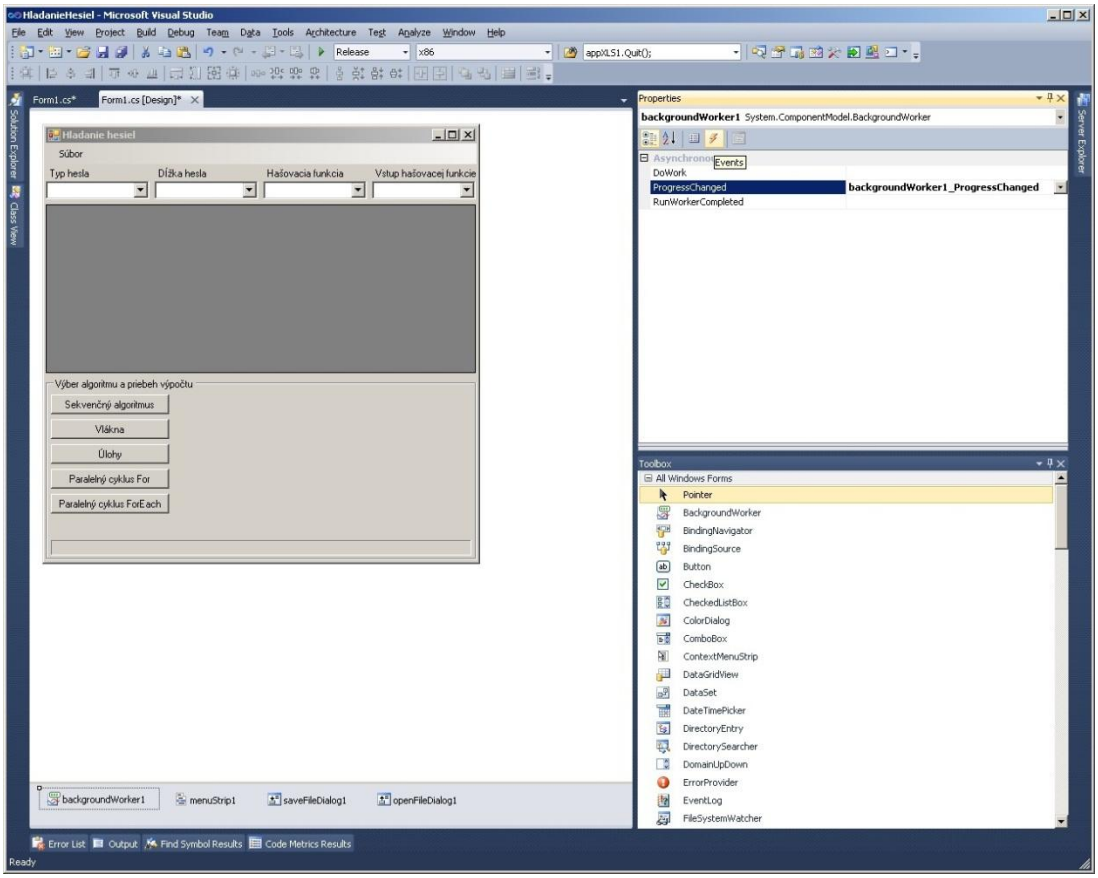
Metóda InicializujBackgroundWorker

Jednou z metód triedy formulára je aj metóda **InicializujBackgroundWorker**, ktorá umožňuje aplikácii adekvátne reagovať na vzniknuté udalosti objektu **backgroundWorker1** v prípade akejkoľvek voľby algoritmu na hľadanie hesiel. Metóda podľa hodnoty svojho formálneho parametra pripraví na použitie objekt **backgroundWorker1**. Časť definície metódy sa nachádza v nasledujúcom fragmente zdrojového kódu, kde zobrazená vetva **case** korešponduje s voľbou sekvenčného algoritmu na hľadanie hesiel:

```
switch (volbaAlgoritmu)
{
	case 0:
		backgroundWorker1.DoWork -= backgroundWorker1SekvencnyAlg_DoWork;
		backgroundWorker1.RunWorkerCompleted -=
		backgroundWorker1SekvencnyAlg_RunWorkerCompleted;
		backgroundWorker1.DoWork -= backgroundWorker1Vlakna_DoWork;
		backgroundWorker1.RunWorkerCompleted -=
		backgroundWorker1Vlakna_RunWorkerCompleted;
		backgroundWorker1.DoWork -= backgroundWorker1Ulohy_DoWork;
		backgroundWorker1.RunWorkerCompleted -=
		backgroundWorker1Ulohy_RunWorkerCompleted;
		backgroundWorker1.DoWork -= backgroundWorker1ParalelnyFor_DoWork;
		backgroundWorker1.RunWorkerCompleted -=
		backgroundWorker1ParalelnyFor_RunWorkerCompleted;
		backgroundWorker1.DoWork -=
		backgroundWorker1ParalelnyForEach_DoWork;
		backgroundWorker1.RunWorkerCompleted -=
		backgroundWorker1ParalelnyForEach_RunWorkerCompleted;

		backgroundWorker1.DoWork += new
			DoWorkEventHandler(backgroundWorker1SekvencnyAlg_DoWork);
		backgroundWorker1.RunWorkerCompleted += new
			RunWorkerCompletedEventHandler(
			backgroundWorker1SekvencnyAlg_RunWorkerCompleted);
		break;
//...
```

Príprava inštancie triedy **BackgroundWorker** na použitie spočíva v odobratí už registrovaných a následnom zaregistrovaní potrebných cieľových metód, ktoré majú byť volané v prípade, ak nastanú rôzne udalosti triedy **BackgroundWorker** pre zvolený algoritmus. Ako si môžeme všimnúť, registrácia cieľových metód v kóde sa realizuje pre udalosti **DoWork** a **RunWorkerCompleted**, keďže iba v prípade týchto dvoch udalostí je potrebná odlišná reakcia na ich generovanie v závislosti od zvoleného typu algoritmu pre hľadanie hesiel. Pretože reakcia na udalosť **ProgressChanged** je rovnaká v prípade všetkých použitých algoritmov, môžeme zodpovedajúcu metódu pre túto udalosť zaregistrovať aj pomocou vizuálneho návrhára grafického používateľského rozhrania.



Obr. 3: Registrácia metódy pre asynchrónnu udalosť **ProgressChanged** triedy **BackgroundWorker** v prostredí vizuálneho návrhára grafického používateľského rozhrania

Súčasťou procesu je automatické generovanie kódu zodpovedajúceho programového príkazu prostredníctvom vývojového prostredia do tela inštančnej metódy **InitializeComponent** v tvare:

```
this.backgroundWorker1.ProgressChanged += new
        System.ComponentModel.ProgressChangedEventHandler(
        this.backgroundWorker1_ProgressChanged);
```

Metóda backgroundWorker1_ProgressChanged

Metóda **backgroundWorker1_ProgressChanged** s nasledujúcou definíciou bude vykonaná v prípade, že v rámci objektu **backgroundWorker1** nastane udalosť **ProgressChanged**:

```
private void backgroundWorker1_ProgressChanged(object sender,    ProgressChangedEventArgs e)
{
        lblStatus.Text = e.ProgressPercentage.ToString() + " %";
        progressBar1.Value = e.ProgressPercentage;
}
```

V tele metódy sú aktualizované zodpovedajúce prvky grafického používateľského rozhrania hodnotou dosiahnutého progresu práve exekvovaného algoritmu hľadania hesiel.

Metóda btnSekvencnyAlg_Click

Implementáciu sekvenčného algoritmu na hľadanie hesiel v triede **Form1** tvoria predovšetkým metódy **btnSekvencnyAlg_Click** na obsluhu udalosti **Click** tlačidla **btnSekvencnyAlg**, **backgroundWorker1SekvencnyAlg_DoWork** a metóda **backgroundWorker1SekvencnyAlg_RunWorkerCompleted**. Definíciu prvej z menovaných metód, zabezpečujúcej štart asynchrónnej operácie hľadania hesiel, predstavuje nasledujúci fragment zdrojového kódu:

```
private void btnSekvencnyAlg_Click(object sender, EventArgs e)
{
	if (NastavDlzkuHesla())
	{
		ZmazHesla();
		ZobrazUdaje();
		if (!backgroundWorker1.IsBusy)
		{
			InicializujBackgroundWorker(0);
			// Štart asynchrónnej operácie.
			backgroundWorker1.RunWorkerAsync();
			lblSekvencnyAlg.Text =
			    "Prebieha hľadanie hesiel sekvenčným algoritmom...";
		}
	}
}
```

Implementácie metód obsluhujúcich udalosti **Click** sú pri ostatných použitých algoritmoch v konečnom dôsledku veľmi podobné. Prvým rozdielom je hodnota odovzdávaná formálnemu parametru funkcie **InicializujBackgroundWorker**, ktorá je pre každý z algoritmov jedinečná. Druhý rozdiel predstavuje zobrazovaná textová notifikácia pre používateľa. Z tohto dôvodu budeme pri opisoch implementácie ostatných algoritmov na hľadanie hesiel metódy na obsluhu udalostí **Click** vynechávať.

Metóda backgroundWorker1SekvencnyAlg_DoWork

Programový príkaz, volajúci v predchádzajúcom fragmente zdrojového kódu metódu **RunWorkerAsync**, spôsobí vznik udalosti, na ktorú objekt **backgroundWorker1** zareaguje volaním metódy **backgroundWorker1SekvencnyAlg_DoWork** s nasledujúcou definíciou:

```
private void backgroundWorker1SekvencnyAlg_DoWork(
	object sender, DoWorkEventArgs e)
{
	sw.Start();
	backgroundWorker1.ReportProgress(0);
	if (volbaFunkcie == 0)
		for (int i = 0; i < md5.Length; i++)
		{
			md5[i].NajdiHeslo();
			backgroundWorker1.ReportProgress((int)((float)(i + 1) /
			(float)md5.Length * 100));
		}
	// Implementácia algoritmu pre hašovaciu funkciu SHA-2 je vynechaná.
	sw.Stop();
	e.Result = sw.Elapsed.TotalSeconds.ToString("0.00");
	sw.Reset();
}
```

Hlavná funkcionalita metódy je koncentrovaná do cyklov **for**, z ktorých je exekvovaný vždy iba jeden, podľa toho, s akou hašovacou funkciou aplikácia práve pracuje. Ak sa bližšie pozrieme na prvý z cyklov, ktorý je zobrazený, tak zistíme, že v každej jeho iterácii je volaná metóda **NajdiHeslo** na inej inštancii triedy **MD5** nachádzajúcej sa v poli s identifikátorom **md5**. Po nájdení príslušného hesla je metóde **ReportProgress** odovzdaná percentuálna hodnota progresu exekúcie cyklu. Okrem uvedeného metóda uskutoční meranie svojho exekučného času.

Metóda backgroundWorker1SekvencnyAlg_RunWorkerCompleted

Po úspešnom ukončení, prípadne zrušení exekúcie metódy **backgroundWorker1SekvencnyAlg_DoWork**, napríklad z dôvodu vzniku chybovej výnimky, je ako reakcia na vznik udalosti **RunWorkerCompleted** volaná metóda **backgroundWorker1SekvencnyAlg_RunWorkerCompleted** s nasledujúcou definíciou:

```
private void backgroundWorker1SekvencnyAlg_RunWorkerCompleted
(object sender, RunWorkerCompletedEventArgs e)
{
	if (e.Error != null)
		MessageBox.Show(e.Error.Message, "Hľadanie hesiel",
			MessageBoxButtons.OK, MessageBoxIcon.Error);
	else
	{
		lblSekvencnyAlg.Text = "Exekučný čas sekvenčného algoritmu: " +
		e.Result.ToString() + " s.";
		ZobrazUdaje();
	}
}
```

V prípade vzniku chybovej výnimky metóda zobrazí používateľovi chybovú správu. Naopak, po úspešnom ukončení exekúcie v metóde **backgroundWorker1SekvencnyAlg_DoWork** zobrazí nájdené heslá a exekučný čas algoritmu. Podobne, ako je to pri metóde **btnSekvencnyAlg_Click**, aj implementácia metódy **backgroundWorker1SekvencnyAlg_RunWorkerCompleted** je veľmi podobná implementáciám použitým pri ostatných algoritmoch hľadania hesiel. Z tohto dôvodu budeme pri opisoch implementácie ostatných algoritmov metódy reagujúce na vznik udalosti **RunWorkerCompleted** vynechávať.

Metóda backgroundWorker1Vlakna_DoWork

Predstavuje jadro implementácie algoritmu využívajúceho pre svoju paralelizáciu inštancie triedy **Thread**. Z hľadiska pochopenia funkcionality implementovanej v metóde je dôležitý nasledujúci fragment jej zdrojového kódu:

```
int aktualnyProgres = 0;
backgroundWorker1.ReportProgress(aktualnyProgres);
if (volbaFunkcie == 0)
	for (int i = 0; i < vlakna.Length; i++)
	{
		if (i < (vlakna.Length - 1))
			vlakna[i] = new Thread((object indexVlakna) =>
			{
				int index = Convert.ToInt32(indexVlakna);
				for (int j = index *
				RozdeleniePrace.MnozstvoPraceVPracovnejJednotke;
```

```
                                j < ((index + 1) *
                                RozdeleniePrace.MnozstvoPraceVPracovnejJednotke; j++)
                        {
                                md5[j].NajdiHeslo();
                                backgroundWorker1.ReportProgress(
                                        (int)((float)Interlocked.Increment(
                                        ref aktualnyProgres) / (float)
                                        RozdeleniePrace.CelkoveMnozstvoPrace * 100));
                        }
                });
        else
                vlakna[i] = new Thread((object indexVlakna) =>
                {
                        int index = Convert.ToInt32(indexVlakna);
                        for (int j = index *
                        RozdeleniePrace.MnozstvoPraceVPracovnejJednotke;
                        j < RozdeleniePrace.CelkoveMnozstvoPrace; j++)
                        {
                                md5[j].NajdiHeslo();
                                backgroundWorker1.ReportProgress(
                                        (int)((float)Interlocked.Increment(
                                        ref aktualnyProgres) / (float)
                                        RozdeleniePrace.CelkoveMnozstvoPrace *
                                        100));
                        }
                });
    }
// Implementácia algoritmu pre hašovaciu funkciu SHA-2 je vynechaná.
for (int i = 0; i < vlakna.Length; i++)
{
    vlakna[i].Start(i);
}
for (int i = 0; i < vlakna.Length; i++)
{
    vlakna[i].Join();
}
```

Ako jediný z algoritmov využíva statickú triedu **RozdeleniePrace** na vytvorenie zodpovedajúceho počtu pracovných jednotiek a na rozdelenie celkového množstva práce medzi pracovné jednotky tak, aby ich zaťaženie bolo rovnomerné, pričom pod pracovnou jednotkou tu rozumieme pracovné programové vlákno. Počet pracovných programových vlákien, ktoré budú vytvorené, je daný dĺžkou jednorozmerného poľa s identifikátorom **vlakna** a zároveň hodnotou dátovej položky **pocetPracovnychJednotiek**.

Programové príkazy, nachádzajúce sa vo vetve **if**, sa vykonajú pre všetky pracovné programové vlákna, okrem v poradí posledného pracovného programového vlákna. Pre posledné pracovné programové vlákno sú vykonané príkazy vo vetve **else**. V každom cykle **for** je teda vytvorená jedna inštancia triedy **Thread**, do ktorej je pomocou vnoreného cyklu **for** injektovaný programový kód na nájdenie zodpovedajúceho počtu hesiel. Vo vetve **if**, nachádzajúcej sa v tele nadradeného cyklu, je metóda **NajdiHeslo** triedy **MD5** volaná na takom počte inštancií, ktorý predstavuje hodnotu dátovej položky **MnozstvoPraceVPracovnejJednotke** statickej triedy **RozdeleniePrace**. Každé z pracovných vlákien pracuje s hodnotou **indexVlakna**, ktorá je pracovnému vláknu odovzdaná pri štarte jeho exekúcie, pričom predstavuje poradie, v akom bola exekúcia na tomto vlákne spustená.

Pracovné programové vlákno vytvorené vo vetve **else** rozhodovacieho príkazu bude volať metódu **NajdiHeslo,** a to na takom počte inštancií triedy **MD5**, ktorý predstavuje rozdiel celkového množstva doteraz uskutočnených volaní na všetkých vytvorených pracovných programových vláknach

a hodnotou dátovej položky **CelkoveMnozstvoPrace** statickej triedy **RozdeleniePrace**. Každé vytvorené pracovné programové vlákno v rámci svojho cyklu **for** po nájdení príslušného hesla odovzdá metóde **ReportProgress** percentuálnu hodnotu progresu exekúcie.

Za povšimnutie stojí použitie metódy **Increment** statickej triedy **Interlocked** na uskutočnenie bezpečnej atomickej inkrementácie celočíselnej hodnoty premennej **aktualnyProgres**. Uvedený algoritmus je zhodný aj v prípade použitia hašovacej funkcie **SHA-2**.

Metóda backgroundWorker1Ulohy_DoWork

Algoritmus využívajúci inštancie triedy **Task** je v triede **Form1** implementovaný metódou **backgroundWorker1Ulohy_DoWork**. Tento je v porovnaní s algoritmom použitým pri práci s programovými vláknami veľmi jednoduchý. Ekvivalent predchádzajúceho fragmentu zdrojového kódu za predpokladu použitia inštancií triedy **Task** je zdrojový kód nachádzajúci sa v tele metódy **backgroundWorker1Ulohy_DoWork**:

```
if (volbaFunkcie == 0)
	for (int i = 0; i < ulohy.Length; i++)
	{
		ulohy[i] = new Task(md5[i].NajdiHeslo);
	}
// Implementácia algoritmu pre hašovaciu funkciu SHA-2 je vynechaná.
for (int i = 0; i < ulohy.Length; i++)
{
	ulohy[i].Start();
}
backgroundWorker1.ReportProgress(0);
for (int i = 0; i < ulohy.Length; i++)
{
	ulohy[i].Wait();
	backgroundWorker1.ReportProgress((int)((float)(i + 1) /
	(float)ulohy.Length * 100));
}
```

Už na prvý pohľad je zrejmá syntaktická jednoduchosť, s akou vytvárame inštancie triedy **Task**. Tieto inštancie sú prvkami jednorozmerného poľa s identifikátorom **ulohy**, ktorého dĺžka zodpovedá hodnote dátovej položky **CelkoveMnozstvoPrace** statickej triedy **RozdeleniePrace**. Každá z inštancií triedy **Task** bude v rámci svojej exekúcie volať metódu **NajdiHeslo** práve na jednej inštancii triedy **MD5** nachádzajúcej sa v poli s identifikátorom **md5**. Reportovanie progresu v hľadaní hesiel sa nachádza v tele cyklu **for**, kde prebieha čakanie na ukončenie exekúcie jednotlivých inštancií triedy **Task**.

Metóda backgroundWorker1ParalelnyFor_DoWork

V procese zjednodušovania syntaxe pri paralelizácii algoritmu hľadania hesiel budeme pokračovať vo fragmente programového kódu metódy **backgroundWorker1ParalelnyFor_DoWork**, ktorá na paralelizáciu svojej činnosti využíva paralelný cyklus **for**:

```
int aktualnyProgres = 0;
backgroundWorker1.ReportProgress(aktualnyProgres);
if (volbaFunkcie == 0)
	Parallel.For(0, RozdeleniePrace.CelkoveMnozstvoPrace, i =>
```

```
        {
                md5[i].NajdiHeslo();
                backgroundWorker1.ReportProgress((int)((float)Interlocked.Increment(
                ref aktualnyProgres) /
                (float)RozdeleniePrace.CelkoveMnozstvoPrace * 100));
        });
// Implementácia algoritmu pre hašovaciu funkciu SHA-2 je vynechaná.
```

V každej z iterácií paralelného cyklu **for** voláme metódu **NajdiHeslo** na jednej inštancii triedy **MD5** z poľa **md5** a zároveň odovzdávame metóde **ReportProgress** volanej v súvislosti s inštanciou **backgroundWorker1** percentuálnu hodnotu progresu exekúcie. Je zrejmé, že v porovnaní s algoritmami využívajúcimi na paralelizáciu svojich činností inštancie tried **Thread** a **Task** dosahujeme vyššiu úroveň abstrakcie a tým aj väčšiu produktivitu práce.

Metóda backgroundWorker1ParalelnyForEach_DoWork

Definícia metódy **backgroundWorker1ParalelnyForEach_DoWork** využívajúca paralelný cyklus **foreach** je podobná definícii metódy **backgroundWorker1ParalelnyFor_DoWork** a jej podstatu tvoria nasledujúce programové príkazy:

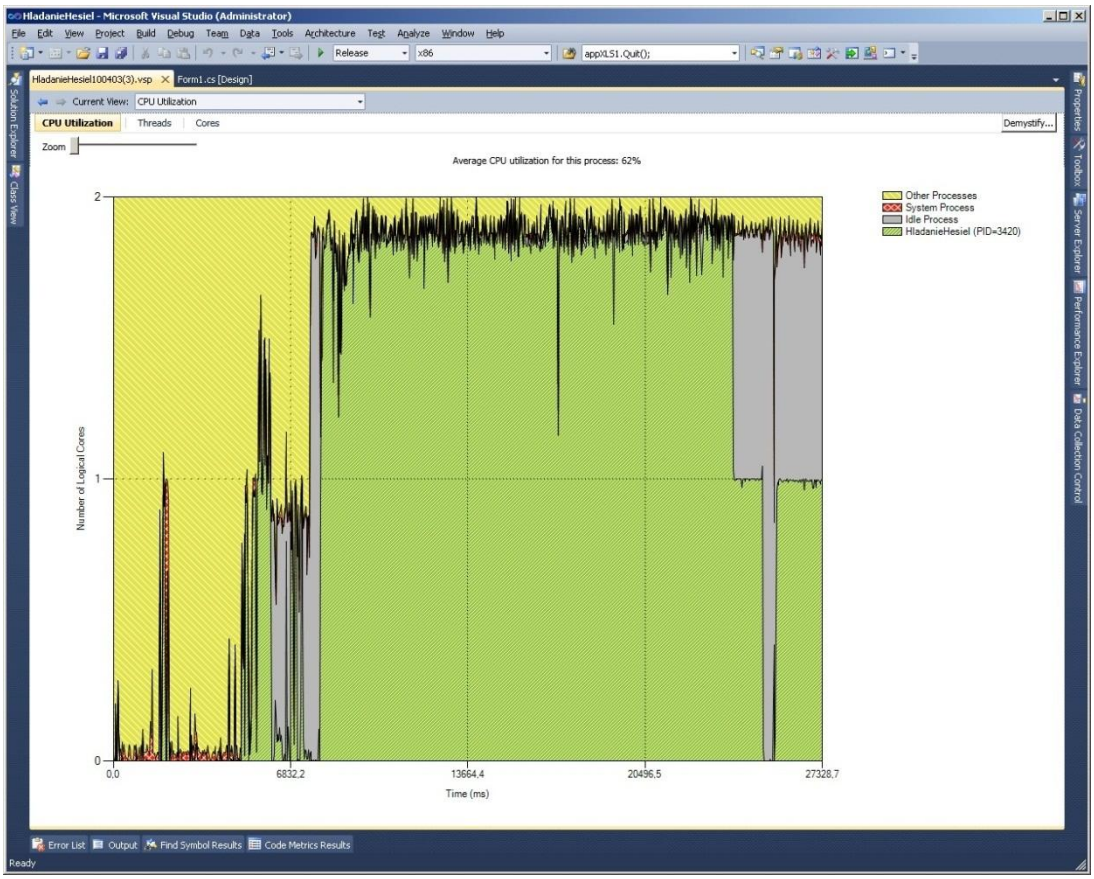
```
int aktualnyProgres = 0;
backgroundWorker1.ReportProgress(aktualnyProgres);
if (volbaFunkcie == 0)
        Parallel.ForEach(md5, instanciaMd5 =>
        {
                instanciaMd5.NajdiHeslo();
                backgroundWorker1.ReportProgress((int)((float)Interlocked.Increment(
                ref aktualnyProgres) /
                (float)RozdeleniePrace.CelkoveMnozstvoPrace * 100));
        });
// Implementácia algoritmu pre hašovaciu funkciu SHA-2 je vynechaná.
```

3 Testovanie paralelnej aplikácie

3.1 Diagnostika paralelnej aplikácie pomocou profilovacieho nástroja

Profilovanie paralelnej aplikácie nám môže pomôcť identifikovať slabé miesta návrhu a implementácie paralelného algoritmu a na základe ich eliminácie môžeme v konečnom dôsledku dospieť k zvýšeniu výkonnosti našej aplikácie.

Prvou skúmanou charakteristikou v reporte **Concurrency Profiling Report** je **Average CPU Utilization** v okne **CPU Utilization**. Táto charakteristika skúma schopnosť aplikácie využiť dostupné exekučné jadrá viacjadrového procesora, a teda schopnosť využiť dostupnú výpočtovú kapacitu počítača.

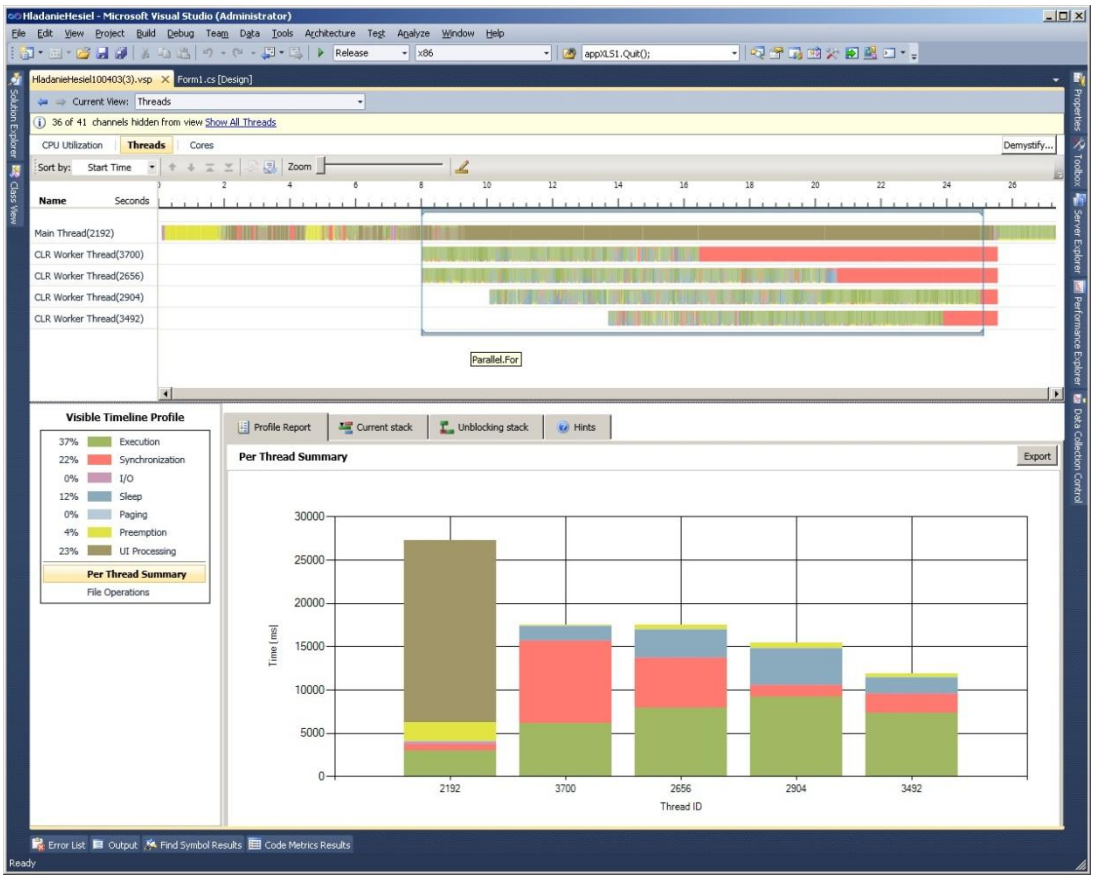


Obr. 4: Profilovanie paralelnej aplikácie – **Concurrency Profiling Report** – okno **CPU Utilization**

Výkaz tvorí graf, ktorý na horizontálnej osi zobrazuje exekučný čas paralelnej aplikácie a na vertikálnej osi exekučné jadrá viacjadrového procesora. Ako môžeme vidieť, počas prvých približne siedmich sekúnd behu našej paralelnej aplikácie, boli exekučné jadrá procesora zaneprázdnené predovšetkým vykonávaním inštrukcií procesov, ktoré sú profilovacím nástrojom označené žltou farbou a identifikátorom **Other Processes**. To zodpovedá času, ktorý používateľ potrebuje po štarte aplikácie na interakciu s grafickým používateľským rozhraním až do momentu samotného spustenia algoritmu hľadania hesiel. Oblasť grafu vyznačená zelenou farbou a identifikátorom procesu **HladanieHesiel** predstavuje čas a intenzitu zaťaženia exekučných jadier viacjadrového procesora exekúciou našej paralelnej aplikácie. Na obrázku môžeme pozorovať interval, v ktorom naša paralelná aplikácia v najväčšej miere zaťažuje obe exekučné jadrá procesora. Tento interval reprezentuje čas exekúcie algoritmu slúžiaceho na hľadanie hesiel.

Podľa reportovanej hodnoty paralelná aplikácia počas celej doby svojho behu využívala v priemere iba 62 % výpočtovej kapacity počítača. Z grafu je však zrejmé, že počas exekúcie samotného algoritmu hľadania hesiel aplikácia využila dostupnú výpočtovú kapacitu v priemere na viac ako 85 %. Čo je prijateľná hodnota, ktorá bola pri návrhu aplikácie očakávaná. Skreslenie reportovanej hodnoty je spôsobené práve interakciou používateľa s grafickým používateľským rozhraním tesne po spustení aplikácie a ukončení exekúcie algoritmu hľadania hesiel.

Ďalšou časťou výkazu je okno **Threads**, ktoré analyzuje stavy jednotlivých programových vlákien počas životného cyklu paralelnej aplikácie.



Obr. 5: Profilovanie paralelnej aplikácie – **Concurrency Profiling Report** – okno **Threads**

V hornej časti výkazu sú zobrazené jednotlivé vlákna testovanej paralelnej aplikácie spolu s dĺžkou ich spracovania v sekundách. Zobrazenie ostatných programových vlákien je filtrované.

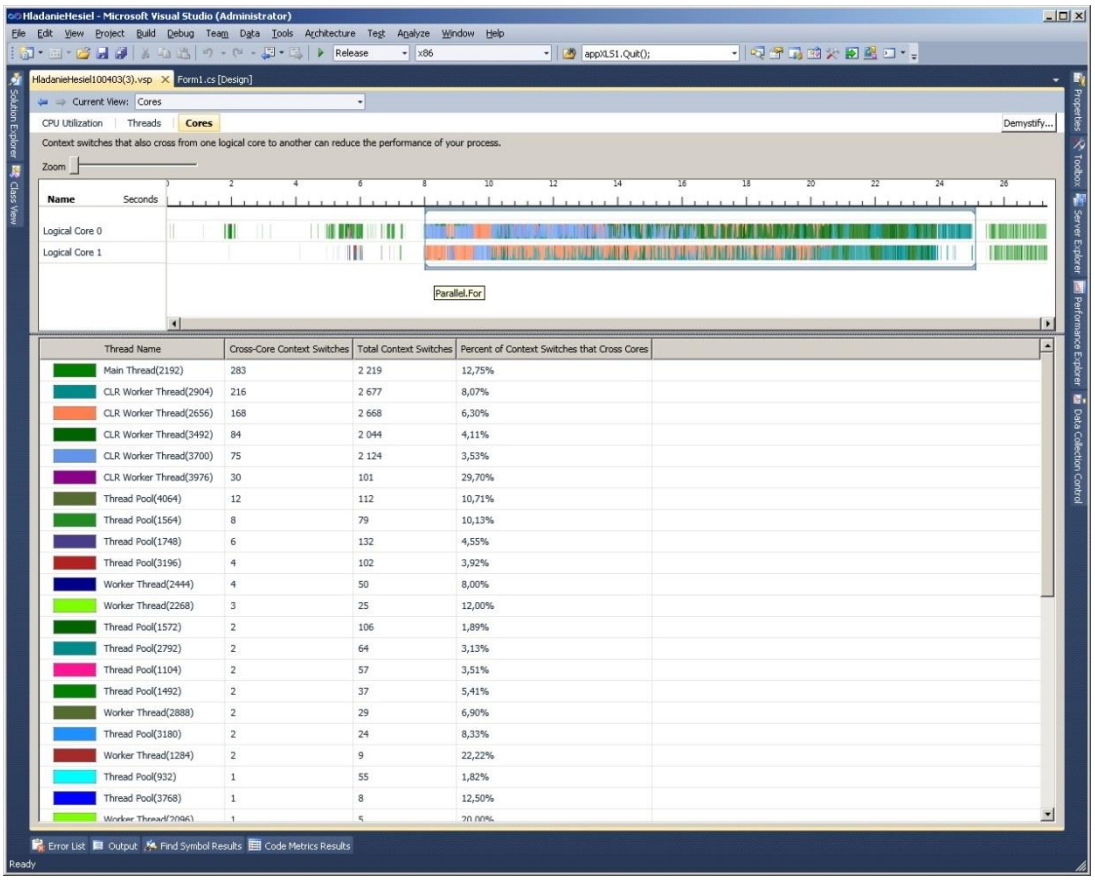
Dĺžka primárneho programového vlákna označeného identifikátorom **Main Thread** predstavuje celú dobu behu paralelnej aplikácie **Hladanie hesiel**. Doba exekúcie paralelného cyklu **for** v aplikácii je označená modrým obdĺžnikom, ktorý zároveň označuje časový interval, v ktorom prebiehala exekúcia algoritmu hľadania hesiel. Ako môžeme pozorovať, plánovač úloh v spolupráci so spoločným behovým prostredím vytvoril pri exekúcii paralelného cyklu **for** pre našu aplikáciu 4 pracovné programové vlákna, medzi ktoré bola rozdelená exekúcia algoritmu hľadania hesiel. Životné cykly týchto programových vlákien boli ukončené spolu s ukončením exekúcie paralelného cyklu. V každom riadku prislúchajúcom k istému programovému vláknu vieme podľa jeho farby určiť, v akom stave sa programové vlákno nachádza. Ideálnym prípadom pri pracovných programových vláknach je stav označený zelenou farbou a identifikátorom **Execution**, počas ktorého vlákno vykonáva svoje programové príkazy.

V spodnej časti okna **Threads** v záložke **Profile Report** sú zobrazené kumulatívne časové charakteristiky jednotlivých programových vlákien počas celej doby trvania ich životných cyklov.

Z pohľadu efektivity exekúcie paralelnej aplikácie na danej testovacej konfigurácii počítača by bolo výhodnejšie vytvorenie iba dvoch pracovných programových vlákien. Toto by znížilo postrannú réžiu súvisiacu so správou programových vlákien, ktorú generuje spoločné behové prostredie.

Poslednou charakteristikou paralelnej aplikácie, ktorú pomocou diagnostického nástroja sledujeme, je migrácia programových vlákien medzi exekučnými jadrami viacjadrového procesora. Migrácia programových vlákien naprieč exekučnými jadrami viacjadrového procesora je vo všeobecnosti nežiaduci jav, ktorý môže mať značne negatívny vplyv na výkonnosť našej paralelnej aplikácie. Súvisí to s potrebou uchovania kontextu vlákna pri premiestnení jeho exekúcie na iné exekučné jadro a tiež s potrebou opätovného načítania údajov do vyrovnávacích pamätí.

Požadované zobrazenie charakteristiky nám poskytne okno **Cores** v **Concurrency Profiling Report**.



Obr. 6: Profilovanie paralelnej aplikácie – **Concurrency Profiling Report** – okno **Cores**

V hornej polovici okna je zobrazený graf s exekučnými jadrami viacjadrového procesora a s časom exekúcie paralelnej aplikácie v sekundách. Spodná časť okna zase zobrazuje tabuľku s programovými vláknami a čiastkovými charakteristikami ich migrácie ako:

- Absolútna zmena kontextu programového vlákna medzi exekučnými jadrami.
- Celková absolútna zmena kontextu programového vlákna.
- Podiel absolútnej zmeny kontextu programového vlákna medzi exekučnými jadrami na celkovej absolútnej zmene kontextu programového vlákna v percentách.

Modrým obdĺžnikom je vyznačený časový interval, v ktorom dochádza k exekúcii paralelného cyklu **for**, a teda algoritmu na hľadanie hesiel. V tomto časovom intervale vieme jednoznačne identifikovať exekúciu štyroch pracovných programových vlákien, ktoré boli zobrazené tiež v okne reportu **Threads**. Z charakteristík podporených grafom môžeme usúdiť, že paralelná aplikácia **Hľadanie hesiel** v prípade

testovaného algoritmu dosahuje nízku migráciu programových vlákien medzi jednotlivými exekučnými jadrami viacjadrového procesora, čo pozitívne vplýva na jej dosahovaný výkon.

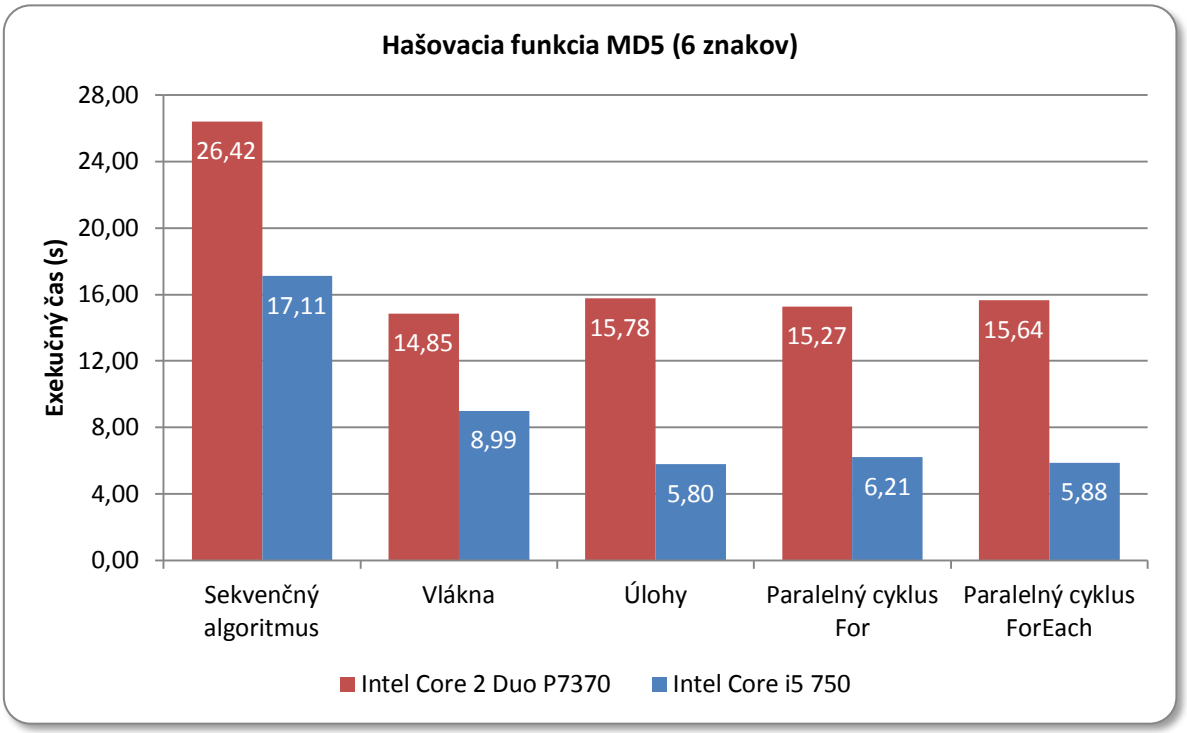
3.2 Meranie výkonnosti paralelnej aplikácie

Výsledky meraní výkonnosti paralelnej aplikácie sú zhrnuté v tab. 1 a graficky na obr. 7 – 8.

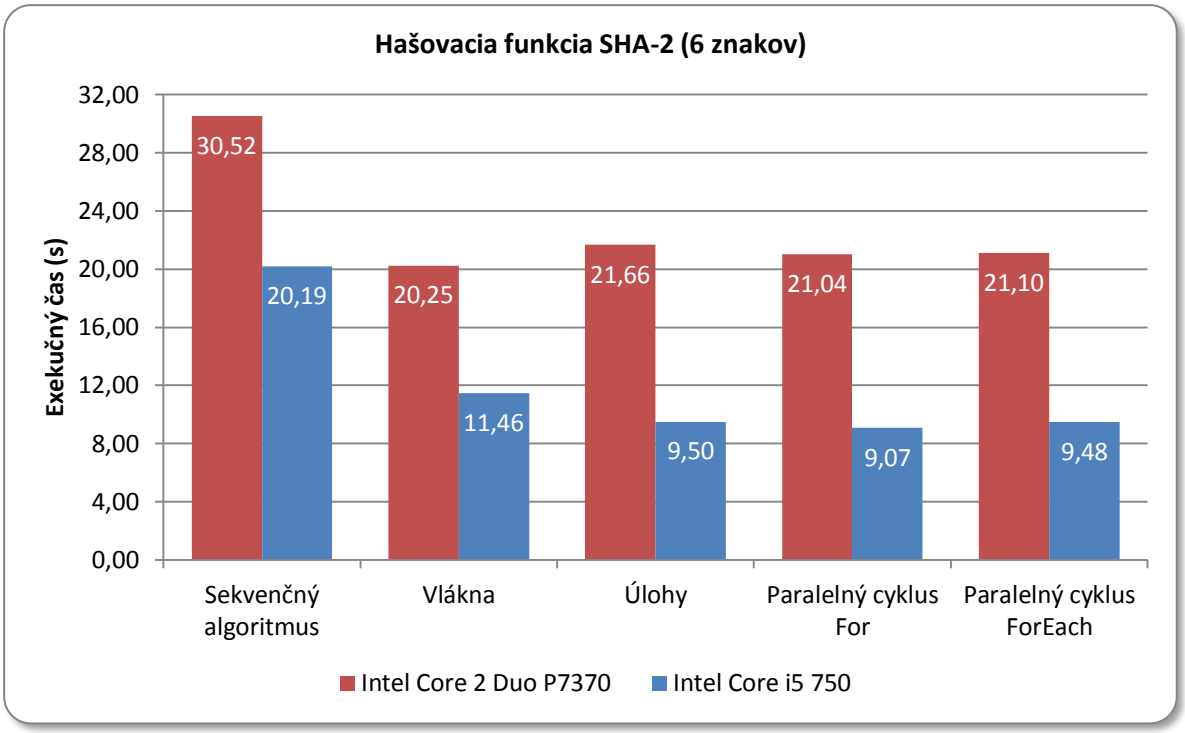
Tab. 1: Výsledky výkonnostných testov paralelnej aplikácie pri vstupnom súbore 6 používateľských mien

Intel Core 2 Duo P7370		**1.**	**2.**	**3.**	**4.**	**5.**	$\overline{E}_T$	N_V	N_V – *klasifikácia*	*e*
MD5 (6 znakov)	Sekvenčný algoritmus	26,37	26,37	26,53	26,65	26,19	26,42	1,00	sublineárny	50,00
	Vlákna	14,43	14,90	14,97	14,97	14,96	14,85	1,78	sublineárny	88,99
	Úlohy	16,08	15,38	16,29	16,12	15,05	15,78	1,67	sublineárny	83,70
	Paralelný cyklus **for**	15,10	15,45	15,11	15,17	15,53	15,27	1,73	sublineárny	86,50
	Paralelný cyklus **foreach**	16,04	15,71	15,35	15,99	15,12	15,64	1,69	sublineárny	84,46
SHA-2 (6 znakov)	Sekvenčný algoritmus	30,50	30,53	30,49	30,53	30,55	30,52	1,00	sublineárny	50,00
	Vlákna	20,24	20,23	20,28	20,33	20,17	20,25	1,51	sublineárny	75,36
	Úlohy	21,59	21,62	21,75	21,64	21,70	21,66	1,41	sublineárny	70,45
	Paralelný cyklus **for**	20,68	20,85	21,23	21,22	21,20	21,04	1,45	sublineárny	72,54
	Paralelný cyklus **foreach**	20,92	21,50	21,18	21,22	20,69	21,10	1,45	sublineárny	72,32
Intel Core i5 750		**1.**	**2.**	**3.**	**4.**	**5.**	$\overline{E}_T$	N_V	N_V – *klasifikácia*	*e*
MD5 (6 znakov)	Sekvenčný algoritmus	17,44	15,55	17,50	17,58	17,50	17,11	1,00	sublineárny	25,00
	Vlákna	9,06	8,94	8,98	8,96	9,03	8,99	1,90	sublineárny	47,57
	Úlohy	5,77	5,69	5,72	5,66	6,14	5,80	2,95	sublineárny	73,82
	Paralelný cyklus **for**	5,72	6,15	6,55	6,10	6,53	6,21	2,76	sublineárny	68,90
	Paralelný cyklus **foreach**	5,69	5,78	5,71	5,72	6,51	5,88	2,91	sublineárny	72,74
SHA-2 (6 znakov)	Sekvenčný algoritmus	20,20	20,20	20,24	20,15	20,17	20,19	1,00	sublineárny	25,00
	Vlákna	11,45	11,48	11,41	11,45	11,51	11,46	1,76	sublineárny	44,05
	Úlohy	8,91	9,57	9,77	9,71	9,56	9,50	2,12	sublineárny	53,11
	Paralelný cyklus **for**	8,72	9,00	9,32	9,18	9,14	9,07	2,23	sublineárny	55,64
	Paralelný cyklus **foreach**	9,19	9,58	9,41	9,57	9,65	9,48	2,13	sublineárny	53,25

1. – 5.	sú exekučné časy (v sekundách) algoritmov v jednotlivých meraniach výkonnosti,
$\bar{E}_T$	je priemerný exekučný čas algoritmu (v sekundách),
N_V	je nárast výkonnosti paralelného algoritmu v porovnaní so sekvenčným algoritmom,
N_V– *klasifikácia*	predstavuje klasifikáciu nárastu výkonnosti paralelného algoritmu v porovnaní so sekvenčným algoritmom,
e	je efektivita využitia výpočtových zdrojov počítačového systému v percentách.



Obr. 7: Porovnanie priemerných exekučných časov jednotlivých algoritmov pri hašovacej funkcii **MD5** na oboch testovacích konfiguráciách počítačov



Obr. 8: Porovnanie priemerných exekučných časov jednotlivých algoritmov pri hašovacej funkcii **SHA-2** na oboch testovacích konfiguráciách počítačov

Ako môžeme vidieť, v prípade testovania algoritmov na prvej konfigurácii počítača, dosahuje pri oboch hašovacích funkciách jednoznačne najlepší exekučný čas algoritmus, využívajúci na paralelizáciu svojich činností inštancie triedy **Thread**. Toto je spôsobené najnižšou mierou abstrakcie tejto techniky, ktorá generuje najnižšiu postrannú réžiu zo strany spoločného behového prostredia, a teda najmenej negatívne ovplyvňuje výkonnosť celého algoritmu.

Vyššia výkonnosť tohto algoritmu je vykúpená najvyššími nárokmi kladenými na vývojára aplikácie spomedzi všetkých implementovaných paralelných algoritmov. V konečnom dôsledku je dosiahnutá nízka produktivita práce vývojára, ktorá súvisí s nízkou mierou abstrakcie pri implementácii algoritmu. Na porovnanie, paralelné algoritmy, ktoré využívajú novinky knižnice **TPL**, sú síce v istých situáciách pomalšie, avšak ich použitím sa stáva aplikácia v každej situácii robustnejšou a lepšie škálovateľnou. Navyše, vývojár paralelnej aplikácie tu dosahuje vysokú produktivitu práce.

Výsledky merania výkonnosti algoritmov na druhej konfigurácii počítača v prípade funkcie **MD5** ukázali v najlepšom svetle algoritmus využívajúci na paralelizáciu svojich činností inštancie triedy **Task** a v prípade funkcie **SHA-2** algoritmus využívajúci paralelný cyklus **for**. Algoritmus využívajúci inštancie triedy **Thread** výkonnostne zaostáva z dôvodu neoptimálnej distribúcie pracovného zaťaženia naprieč vytvorenými programovými vláknami, ktoré zabezpečuje statická trieda **RozdelenePrace**. Ukázalo sa teda, že trieda **RozdelenePrace** implementujúca distribúciu pracovného zaťaženia nepracuje za každých okolností optimálne. To potvrdzuje jednoznačný prínos a prednosti, ktoré knižnica **TPL** komerčným vývojárom ponúka.

Celkovo nižšie úrovne nárastu výkonnosti a efektivity sú dosahované pri testovaní algoritmov na druhej konfigurácii počítača so štyrmi exekučnými jadrami v porovnaní s prvou konfiguráciou, kde pracuje

procesor s dvomi exekučnými jadrami. Je to do istej miery spôsobené spomenutou neoptimálnou distribúciou pracovného zaťaženia pri tak malom počte pracovných jednotiek v podobe hľadaných hesiel a väčšom počte exekučných jadier. Tento jav môžeme pozorovať či už v prípade použitia algoritmu s inštanciami triedy **Thread**, ale aj v prípade algoritmov využívajúcich programové konštrukcie knižnice **TPL**. Druhým dôvodom pravdepodobne bude vyššia pracovná frekvencia procesora Intel Core i5 750 pri zaťažení jedného exekučného jadra, ako pri súčasnom zaťažení všetkých jeho exekučných jadier.

Použitá literatúra

1. ALBAHARI, J., ALBAHARI, B. 2010. *C# 4.0 in a Nutschell.* Sebastopol: O´Reilly Media, 2010. 1033 s. ISBN 978-0-596-80095-6.
2. DUFFY, J. 2009. *Concurrent Programming on Windows.* Boston: Addison-Wesley Pearson Education, 2009. 958 s. ISBN 978-0-321-43482-1.
3. HANÁK, J. 2006. *C# praktické příklady.* Praha: Grada Publishing, 2006. 288 s. ISBN 80-247-0988-0.
4. HANÁK, J. 2008. *Objektovo orientované programovanie v jazyku C# 3.0.* Brno: Artax, 2008. 124 s. ISBN 978-80-870-17-02-9.
5. HANÁK, J. 2009. *Praktické paralelné programovanie v jazykoch C# 4.0 a C++.* Brno: Artax, 2009. 132 s. ISBN 978-80-87017-06-7.
6. HANÁK, J. 2009. *Základy paralelného programovania v jazyku C# 3.0.* Brno: Artax, 2009. 201 s. ISBN 978-80-87017-03-6.
7. HILYARD, J., TAILHET, S. 2008. *C# 3.0 Cookbook.* Sebastopol: O´Reilly Media, 2008. 857 s. ISBN 978-0-596-51610-9.

Ing. Vladimír Juhás
je absolventom študijného programu Hospodárska informatika na Fakulte hospodárskej informatiky Ekonomickej univerzity v Bratislave (FHI EU). Pracuje ako softvérový vývojár, pričom sa venuje najmä implementácii podnikového informačného systému Microsoft Dynamics NAV a vývoju webovských a desktopových aplikácií.

OD študentov
PRE študentov