Microsoft

Ján Hanák

Základy paralelného programovania v jazyku C# 3.0

Microsoft® Visual Studio® 2008

Ján Hanák

Základy paralelného programovania v jazyku C# 3.0

Artax
2009

Autor: Ing. Ján Hanák, MVP

Základy paralelného programovania v jazyku C# 3.0

Recenzenti:	doc. RNDr. Jozef Fecenko, CSc. Ing. Ľudovít Markus, CSc.
Vydanie:	prvé
Rok prvého vydania:	2009
Náklad:	150 ks
Jazyková korektúra:	Ing. Peter Kubica
Vydal:	Artax a.s., Žabovřeská 16, 616 00 Brno pre Microsoft s.r.o., Vyskočilova 1461/2a, 140 00 Praha 4
Tlač:	Artax a.s., Žabovřeská 16, 616 00 Brno
ISBN:	978-80-87017-03-6

Učebný text *Základy paralelného programovania v jazyku C# 3.0* bol schválený v Edičnom pláne Ekonomickej univerzity v Bratislave na rok 2009 ako vysokoškolská učebnica pre študentov povinného predmetu *Automatizácia programovania* v študijnom programe *Hospodárska informatika*, 2. stupeň (inžinierske štúdium).

Obsah

Úvod

S príchodom viacjadrových procesorov sa pred vývojármi, programátormi a softvérovými expertmi otvoril nový svet možností vývoja moderného počítačového softvéru. Ako tvorcovia počítačových systémov sa už nemôžeme spoliehať na neustále zvyšovanie sa taktovacej frekvencie procesorov, ktoré počas predchádzajúcich rokov zabezpečovalo implicitný nárast výkonnosti našich softvérových produktov. Perspektíva je zreteľná, pretože počet exekučných jadier na procesoroch sa bude aj v budúcnosti rýchlo množiť. Ak chceme využiť synergický efekt, ktorý generuje súčasná „viacjadrová hardvérová revolúcia“, musíme sa naučiť, ako zhotovovať paralelné programy. Teda programy, ktoré sú schopné súbežne vykonávať viacero úloh, pričom dokážu flexibilne využívať všetku disponibilnú výpočtovú kapacitu hardvérovej platformy.

Cieľom predkladanej vysokoškolskej učebnice je poskytnúť základný teoreticko-praktický výučbový kurz paralelného programovania v jazyku C# 3.0 na vývojovo-exekučnej platforme Microsoft .NET Framework 3.5. Paralelné programovanie je jednou z paradigiem, ktorých zvládnutie je nutné na to, aby sme mohli úspešne čeliť výzvam budúcich technologických progresov. Hoci bola táto vysokoškolská učebnica primárne projektovaná pre potreby študentov 2. stupňa vysokoškolského (inžinierskeho) štúdia odboru *Hospodárska informatika* na *Fakulte hospodárskej informatiky Ekonomickej univerzity v Bratislave*, sme presvedčení o tom, že rovnako dobre poslúži aj študentom na iných vysokých školách informatického zamerania. Prirodzene, poznatky uvádzané v tejto publikácii s výhodou využijú i komerční vývojári, programátori a softvéroví experti, ktorých poslaním je analyzovať, navrhovať a implementovať paralelné počítačové aplikácie.

Napriek tomu, že počítačové vedy poznajú mnoho praktických modelových prístupov k paralelnému programovaniu, rozhodli sme sa koncentrovať na paralelné programovanie v jednom z najmodernejších programovacích jazykov, C# 3.0 od spoločnosti Microsoft. Naša voľba bola podmienená viacerými faktormi.

Po prvé, poslucháči odboru *Hospodárska informatika* na *Fakulte hospodárskej informatiky Ekonomickej univerzity v Bratislave* absolvujú počas 1. stupňa svojho vysokoškolského (bakalárskeho) štúdia výučbové kurzy programovacích jazykov C a C++, v rámci ktorých sa zoznamujú so štruktúrovanou a objektovo orientovanou filozofiou vývoja počítačového softvéru. Keďže chceme, aby študenti rozvíjali svoje nadanie v triáde C → C++ → C#, je príklon k jazyku C# logickým vyústením našich snáh o priame prepojenie pedagogického procesu s potrebami praxe.

Po druhé, pri paralelnom programovaní v jazyku C# 3.0 sú študenti schopní maximalizovať svoju pracovnú produktivitu, pretože vývojovo-exekučná platforma Microsoft .NET Framework 3.5 obsahuje nielen virtuálny exekučný systém, ktorý riadi životné cykly vytvorených aplikácií, ale aj robustnú bázovú knižnicu tried, implementujúcu niekoľkotisícovú množinu dátových typov, ktoré sú charakteristické vysokou mierou abstrakcie.

Po tretie, domnievame sa, že znalosti objektovo orientovaného programovania (OOP) a paralelného objektovo orientovaného programovania (POOP) prispejú k maximalizácii pracovného uplatnenia študentov po skončení 2. stupňa vysokoškolského štúdia. Samozrejme, veľmi radi uvítame, ak sa študenti rozhodnú ďalej prehlbovať svoje informatické vedomosti aj v rámci 3. stupňa vysokoškolského (doktorandského) štúdia. Sme presvedčení o tom, že ak schopní mladí ľudia ovládnu sofistikované informačné technológie, ich potenciál je nekonečný.

Vysokoškolská učebnica *Základy paralelného programovania v jazyku C# 3.0* je podľa našich zistení jedinou slovenskou publikáciou, ktorá poskytuje elementárny výučbový kurz paralelného programovania pomocou najmodernejších technológií spoločnosti Microsoft.

Autor knihy by si rád splnil svoju milú povinnosť a srdečne poďakoval recenzentom, doc. RNDr. Jozefovi Fecenkovi, CSc. a Ing. Ľudovítovi Markusovi, CSc. za dôkladné preštudovanie diela a námety na jeho ďalšie skvalitnenie. Rovnako úprimne ďakuje autor Ing. Jiřímu Burianovi a Mgr. Miroslavovi Kubovčíkovi zo spoločnosti Microsoft za ich mnohoročnú aktívnu spoluprácu, ktorej výsledkom sú hodnotné produkty

permanentne zlepšujúce stav akademického a vývojárskeho ekosystému. V neposlednom rade vyjadruje autor hlbokú úctu a poďakovanie všetkým nadaným a aktívnym študentkám a študentom, ktorých usilovná práca a ľudský prístup pomáhajú zvyšovať kvalitu pedagogického procesu.

Ján Hanák

Bratislava, február 2009

Obsahová štruktúry knihy

Vysokoškolskú učebnicu tvorí dovedna 18 kapitol s nasledujúcim tematickým zameraním:

1. kapitola: Kapitola podáva výklad hlavných komponentov, z ktorých sa skladá vývojovo-exekučná platforma Microsoft .NET Framework 3.5. Charakterizovaný je virtuálny exekučný systém, bázová knižnica tried, spoločný typový systém a spoločná jazyková špecifikácia.

2. kapitola: Kapitola charakterizuje zostavenie riadenej aplikácie (aplikácie .NET) a ozrejmuje jej internú štruktúru.

3. kapitola: Kapitola sa sústreďuje na technickú analýzu riadenej exekúcie jednovláknovej riadenej aplikácie.

4. kapitola: Kapitola sa koncentruje na technickú analýzu riadenej exekúcie viacvláknovej riadenej aplikácie.

5. kapitola: Kapitola predstavuje základnú typológiu vlákien, pričom charakterizuje programové (aplikačné) vlákna, vlákna jadra operačného systému a hardvérové vlákna.

6. kapitola: Kapitola prezentuje technické podrobnosti hlavných typov procesorových architektúr súčasných počítačových systémov. Výklad sa venuje nasledujúcim typom procesorov: jednojadrové procesory architektúry Intel IA-32 bez a s technológiou HT, viacjadrové procesory architektúry Intel IA-32/Intel 64 bez a s technológiou HT a viacprocesorové architektúry Intel IA-32/Intel 64.

7. kapitola: Kapitola poukazuje na zmenu paradigmy z objektovo orientovaného programovania (OOP) na paralelné objektovo orientované programovanie (POOP).

8. kapitola: Kapitola sa zaoberá kvantifikáciou nárastu výkonnosti počítačových aplikácií pri použití paralelného objektovo orientovaného programovania.

9. kapitola: Kapitola charakterizuje Amdahlov zákon, pričom poukazuje na dôsledky plynúce z jeho praktického použitia.

10. kapitola: Kapitola zoznamuje čitateľov s Gustafsonovým zákonom a umožňuje zaujať iný pohľad na kvantifikáciu nárastu výkonnosti počítačových aplikácií pri paralelnom objektovo orientovanom programovaní.

11. kapitola: Kapitola definuje a vizuálne analyzuje lineárny, sublineárny a superlineárny nárast výkonnosti pri paralelnom objektovo orientovanom programovaní.

12. kapitola: Kapitola sa tematicky orientuje na praktické paralelné programovanie v jazyku C# 3.0. Čitatelia sa dozvedia, ako sa v tomto programovacom jazyku vytvárajú pracovné vlákna a ako môžeme s týmito vláknami manipulovať.

13. kapitola: Kapitola vyzdvihuje použitie fondu pracovných programových vlákien s algoritmizáciou praktického problému.

14. kapitola: Kapitola sa venuje problematike synchronizácie programových vlákien a vysvetleniu vybraných synchronizačných primitív.

15. kapitola: Kapitola ponúka výklad variantov paralelizmu (implicitný a explicitný paralelizmus, dátový paralelizmus, úlohový paralelizmus, paralelizmus dátových tokov, deklaratívny a imperatívny paralelizmus).

16. kapitola: Kapitolu tvorí praktické cvičenie, v ktorom je demonštrovaný proces paralelizácie pôvodne sekvenčnej riadenej aplikácie.

17. kapitola: Kapitola zoznamuje čitateľov s paralelnou platformou Microsoft Parallel Extensions.

18. kapitola: Kapitola ponúka teoreticko-praktický pohľad na natívne rozhranie OpenMP, ktoré slúži na podporu paralelizácie výpočtových procesov.

Typografické konvencie

Aby sme vám čítanie tejto knihy spríjemnili v čo možno najväčšej miere, bol prijatý kódex typografických konvencií, pomocou ktorých došlo k štandardizácii a unifikácii použitých textových štýlov a grafických symbolov. Veríme, že prijaté konvencie zvýšia prehľadnosť a používateľskú prívetivosť výkladu. Prehľad použitých typografických konvencií uvádzame v tab. A.

Typografická konvencia	Ukážka použitia typografickej konvencie
Štandardný text výkladu, ktorý neoznačuje zdrojový kód, identifikátory, modifikátory a kľúčové slová jazyka C# 3.0, ani názvy iných syntaktických elementov a entít, je formátovaný týmto typom písma.	Vývojovo-exekučná platforma Microsoft .NET Framework 3.5 kreuje spoločne s jazykom C# 3.0 jednotnú technologickú bázu na vytváranie moderných riadených aplikácií pre Windows, web a inteligentné mobilné zariadenia.

Tab. A: Prehľad použitých typografických konvencií

Typografická konvencia	Ukážka použitia typografickej konvencie
Názvy ponúk, položiek ponúk, ovládacích prvkov, komponentov, dialógových okien, podporných softvérových nástrojov, typov projektov ako aj názvy ďalších súčastí grafického používateľského rozhrania sú formátované **tučným** písmom. **Tučným** písmom sú rovnako formátované aj identifikátory programových entít, ktoré sú uvádzané priamo vo výkladovom texte.	Nový projekt štandardnej aplikácie pre systém Windows (**Windows Forms Application**) v prostredí produktu Visual C# 2008 založíme takto: 1. Otvoríme ponuku **File**, ukážeme na položku **New** a klikneme na príkaz **Project**. 2. V dialógovom okne **New Project** klikneme v stromovej štruktúre **Project Types** na položku **Visual C#**. 3. Zo súpravy projektových šablón (**Templates**) vyberieme ikonu šablóny **Windows Forms Application**. 4. Do textového poľa **Name** zapíšeme názov pre novú aplikáciu a stlačíme tlačidlo **OK**.

Fragmenty zdrojového kódu jazyka C# 3.0, prípadne akýchkoľvek iných programovacích jazykov sú formátované neproporcionálnym písmom `Courier New`.	`// Definícia premennej typu`
	`// string.`
	`string správa;`
	`// Inicializácia premennej.`
	`správa = "Vitajte v jazyku " +`
	`"C# 3.0!";`
	`// Zobrazenie okna so správou`
	`// pomocou metódy Show triedy`
	`// MessageBox.`
	`MessageBox.Show(správa);`

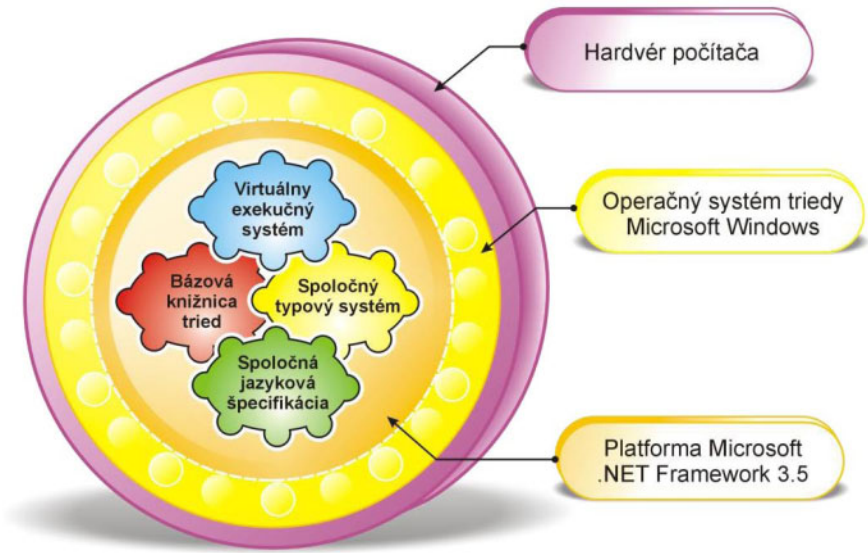
Tab. A: Prehľad použitých typografických konvencií (pokračovanie)

1 Architektonická štruktúra vývojovo-exekučnej platformy Microsoft .NET Framework 3.5

Vývojovo-exekučná platforma Microsoft .NET Framework 3.5 obsahuje kolekciu nasledujúcich základných (jadrových) komponentov:

1. Virtuálny exekučný systém (Virtual Execution System, VES).
2. Bázová knižnica tried (Base Class Library, BCL[1]).
3. Spoločný typový systém (Common Type System, CTS).
4. Spoločná jazyková špecifikácia (Common Language Specification, CLS).

Komponenty vývojovo-exekučnej platformy Microsoft .NET Framework 3.5 sú znázornené na obr. 1.1.



Obr. 1.1: Základné komponenty vývojovo-exekučnej platformy Microsoft .NET Framework 3.5

[1] Pre označenie bázovej knižnice tried sa niekedy používa aj označenie FCL, teda „Framework Class Library“.

V nasledujúcom texte ponúkneme stručnú charakteristiku spomenutých základných komponentov vývojovo-exekučnej platformy Microsoft .NET Framework 3.5.

1.1 Virtuálny exekučný systém (VES)

Virtuálny exekučný systém je softvérový stroj, ktorý realizuje riadenú exekúciu aplikácií .NET. Prijmime dohovor, že termíny „aplikácia .NET“ a „riadená aplikácia“ budú reprezentovať akýkoľvek typový počítačový program, ktorý vyhovuje týmto požiadavkám:

- Program bol vytvorený v niektorom z .NET-kompatibilných programovacích jazykov.
- Proces spustenia a vykonania programu na počítači je riadený virtuálnym exekučným systémom.

Keďže virtuálny exekučný systém vytvára prostredie pre beh ľubovoľnej riadenej aplikácie, je zrejmé, že medzi ním a riadenou aplikáciou existujú veľmi úzke väzby. Spojenie medzi uvedenými entitami je dokonca tak tesné, že bez virtuálneho exekučného systému nie je možné vykonanie aplikácie .NET na počítačovej stanici.

Zatiaľ čo termín „virtuálny exekučný systém“ je v oblasti počítačových vied všeobecným pomenovaním softvérového stroja, ktorý uskutočňuje riadenú exekúciu počítačového programu, spoločnosť Microsoft ako tvorca vývojovo-exekučnej platformy .NET Framework 3.5 označuje svoju vlastnú, teda konkrétnu implementáciu virtuálneho exekučného systému, ako „spoločné behové prostredie CLR[2]“. Hoci v bežných praktických podmienkach sa termíny „virtuálny exekučný systém“ a „spoločné behové prostredie“ vzájomne zamieňajú, považujeme za dôležité poukázať na skutočnosť, že „spoločné behové prostredie“ je len jednou z potenciálnych konkrétnych implementácií „virtuálneho exekučného systému“.

[2] Akronym CLR je skratkou viacslovného pomenovania „Common Language Runtime“.

V záujme zachovania čistoty výkladového textu budeme ďalej v tejto publikácii používať termín „virtuálny exekučný systém“.

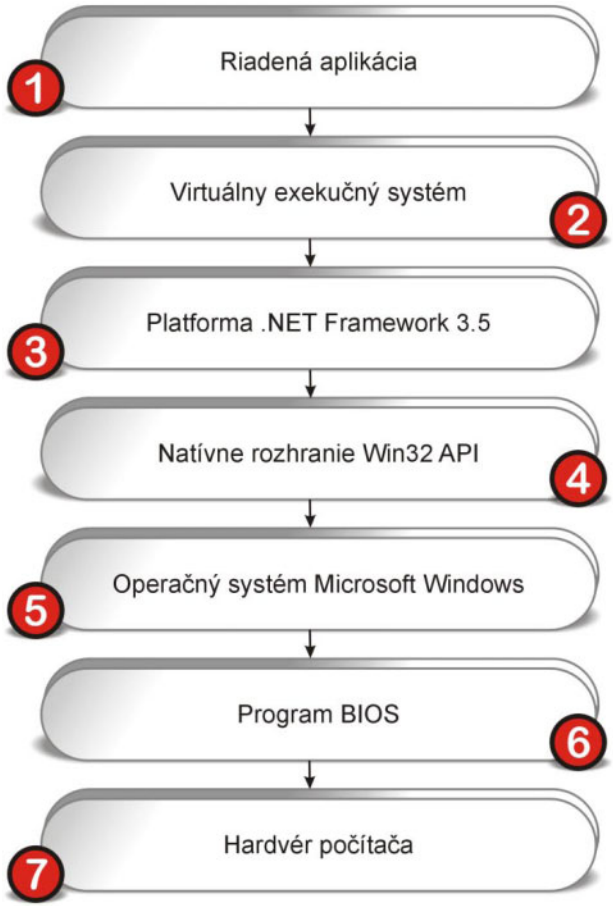
Virtuálny exekučný systém poskytuje riadeným aplikáciám súpravu nízkoúrovňových softvérových služieb, do ktorej patrí:

- Just-In-Time (JIT) kompilácia pseudostrojového kódu MSIL (Microsoft Intermediate Language[3]) do podoby natívnych (strojových) inštrukcií triedy x86.
- Alokácia a dealokácia pamäťových segmentov, ktoré sú asociované s objektmi.
- Garancia typovej bezpečnosti.
- Správa aplikačných domén a programových vlákien.
- Automatický manažment životných cyklov objektov, ktorý zabezpečuje automatický správca pamäte (Garbage Collector, GC).
- Interoperabilita medzi vrstvami natívneho a riadeného programového kódu.

Virtuálny exekučný systém vykonáva svoju činnosť „nad“ vrstvou, ktorú tvorí operačný systém triedy Microsoft Windows[4]. Vizualizáciu komunikačného modelu medzi riadenou aplikáciou, virtuálnym exekučným systémom, operačným systémom Microsoft Windows a hardvérom počítača zobrazuje obr. 1.2.

[3] Jazyk MSIL sa niekedy označuje len ako IL (Intermediate Language), resp. CIL (Common Intermediate Language).

[4] Napriek tomu, že virtuálny exekučný systém, podobne ako aj ďalšie základné komponenty vývojovo-exekučnej platformy Microsoft .NET Framework 3.5, boli úspešne prenesené na iné počítačové platformy a operačné systémy (napríklad Mac OS či Linux), budeme v tejto publikácii uvažovať iba o operačných systémoch triedy Microsoft Windows.



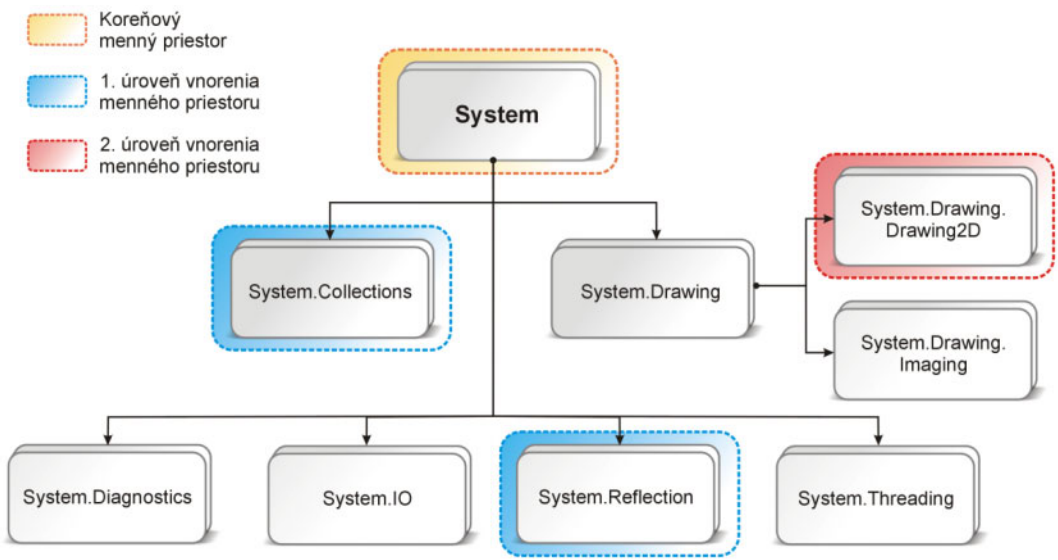
Obr. 1.2: Komunikačný model medzi riadenou aplikáciou, virtuálnym exekučným systémom, operačným systémom a hardvérom počítača

1.2 Bázová knižnica tried (BCL)

Bázová knižnica tried je abstraktnou knižnicou, ktorá bola vytvorená v súlade so smernicami pre hladkú implementáciu objektovo orientovanej a komponentovej paradigmy programovania. Bázová knižnica tried je množinou tisícov hodnotových a odkazových dátových typov, ku ktorým patria štruktúry, vymenované typy, triedy,

rozhrania a delegáti. Dátové typy zapuzdrené do bázovej knižnice tried disponujú vopred naprogramovanou funkcionalitou. Možnosť ich okamžitého použitia efektívne minimalizuje čas, ktorý by bolo nutné vynaložiť na vytvorenie adekvátneho dátového typu v réžii finálneho vývojára. Okrem rýchlej technologickej adopcie je konkurenčnou výhodou bázovej knižnice tried rovnako aj zvýšenie pracovnej produktivity vývojára a v neposlednom rade tiež poskytnutie vysokej miery abstrakcie pri práci s objektmi a spriaznenými dátovými štruktúrami.

Všetky hodnotové a odkazové dátové typy deklarované v bázovej knižnici tried sú na základe svojho pracovného zaradenia roztriedené do fyzicko-logických celkov, tzv. menných priestorov. Koreňový menný priestor má názov **System**, pričom združuje všetky základné menné priestory. Z uvedeného je zrejmé, že menné priestory smú byť vnárané do seba. (Technika vnárania menných priestorov znamená umiestnenie jedného menného priestoru do tela iného menného priestoru.) Keďže hĺbka vnárania menných priestorov môže byť variabilná, smieme pre lepšiu názornosť usporiadanie menných priestorov graficky znázorniť pomocou stromových štruktúr (obr. 1.3).



Obr. 1.3: Vybrané menné priestory bázovej knižnice tried

Menné priestory, ktoré sú vnorené v koreňovom mennom priestore **System**, resp. menné priestory, ktoré sú vnorené vo vnorených menných priestoroch, sú dosiahnuteľné aplikáciou operátora priameho prístupu (tiež bodkového operátora) (.). S nárastom úrovní vnárania sa zvyšuje absolútna početnosť výskytu bodkových operátorov v identifikačných výrazoch, pomocou ktorých sú pre nás požadované menné priestory dosiahnuteľné. Napríklad:

- **System.Collections**, **System.Drawing**, **System.Diagnostics**, **System.Reflection** a **System.Threading** (jedna úroveň vnorenia),

- **System.Drawing.Drawing2D**, **System.Drawing.Imaging**, **System.Data.OleDb**, **System.Linq.Expressions** a **System.Security.Cryptography** (dve úrovne vnorenia).

Bázová knižnica tried je rozšíriteľná, čo znamená, že ak to povaha zabudovaných dátových typov umožňuje, môžeme na ich základe vytvárať nové, odvodené, používateľsky deklarované dátové typy. Najväčšou mierou rozšíriteľnosti disponujú triedy a rozhrania.

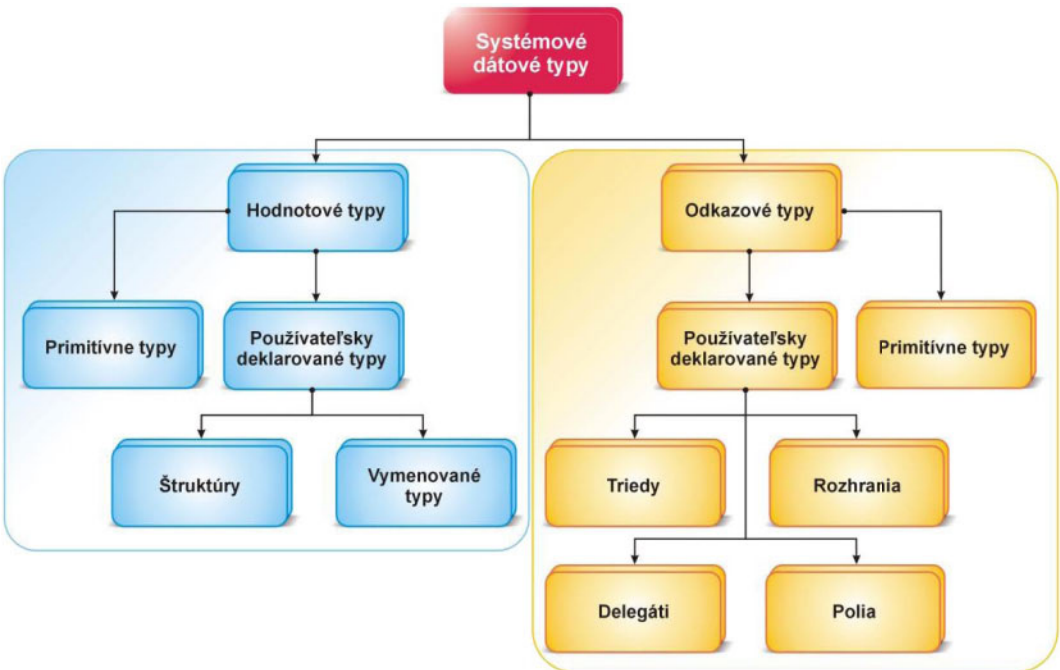
1.3 Spoločný typový systém (CTS)

Spoločný typový systém definuje požiadavky, ktoré musia spĺňať všetky dátové typy, s ktorými vývojári na platforme Microsoft .NET Framework 3.5 pracujú. Spoločný typový systém rozdeľuje dátové typy do dvoch základných skupín:

1. Hodnotové dátové typy.
2. Odkazové dátové typy.

Každá z uvedených množín dátových typov je tvorená primitívnymi typmi a používateľsky deklarovanými typmi (obr. 1.4). Primitívne typy sú dátové typy priamo vstavané do spoločného typového systému, pričom jazykové špecifikácie .NET-kompatibilných programovacích jazykov spravidla definujú špeciálne kľúčové

slová, prostredníctvom ktorých sú tieto primitívne typy explicitne dosiahnuteľné. Používateľsky deklarované dátové typy sú produktom abstraktného myslenia vývojárov, programátorov a softvérových expertov. Tieto dátové typy nie sú vstavané do spoločného typového systému, no spoločný typový systém garantuje podmienky, za akých smú byť uvedené typy vytvárané a používané.



Obr. 1.4: Klasifikácia dátových typov spoločného typového systému

Dátové typy, ktoré definuje spoločný typový systém, sú známe ako systémové dátové typy. Primitívne typy .NET-kompatibilných programovacích jazykov spolupracujú s ekvivalentnými systémovými dátovými typmi. Napríklad, v jazyku C# 3.0 existuje kľúčové slovo **int**, ktoré na syntaktickej úrovni reprezentuje primitívny integrálny znamienkový dátový typ jazyka C# 3.0, ktorý je schopný interpretovať 32-bitové celočíselné hodnoty. V skutočnosti je však kľúčové slovo **int** iba iným pomenovaním (tzv. aliasom) pre príslušný systémový dátový typ **System.Int32**. Presnejšie, primitívny typ **int** jazyka C# 3.0 odkazuje na hodnotovú štruktúru **Int32**,

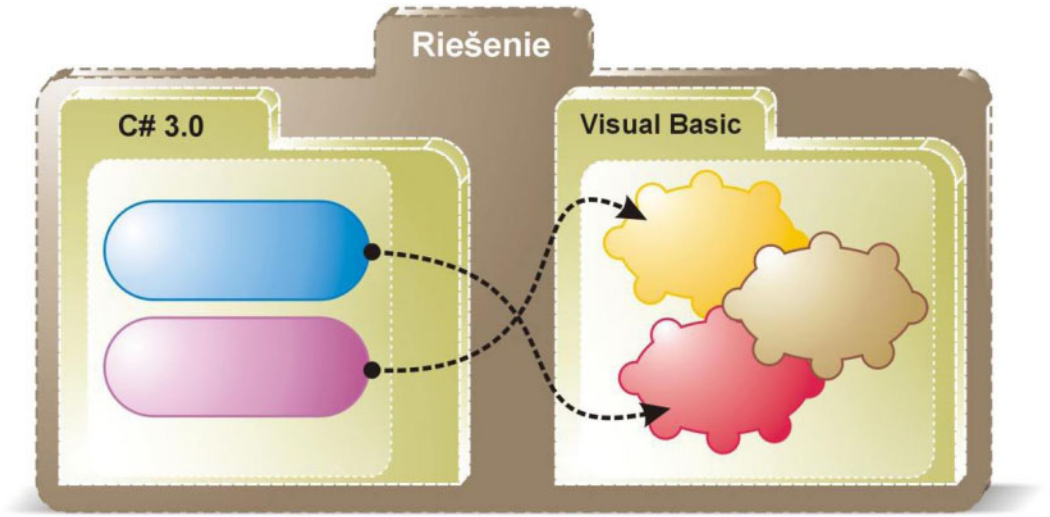
ktorej deklarácia je umiestnená v koreňovom mennom priestore **System**. Analogické interakcie jestvujú aj medzi ostatnými primitívnymi hodnotovými dátovými typmi jazyka C# 3.0 (tab. 1.1).

Primitívny typ jazyka C# 3.0	Ekvivalentný systémový typ	Rozsah hodnôt	Charakteristika
bool	System.Boolean	logická pravda (true), logická nepravda (false)	8-bitová logická hodnota
char	System.Char	znaky súpravy Unicode (interne 16-bitové celé čísla bez znamienka)	16-bitový číselný kód textového znaku súpravy Unicode
short	System.Int16	<-32768, 32767>	16-bitová celočíselná hodnota so znamienkom
ushort	System.UInt16	<0, 65535>	16-bitová celočíselná hodnota bez znamienka
int	System.Int32	$<-2^{31}, 2^{31} - 1>$	32-bitová celočíselná hodnota so znamienkom
uint	System.UInt32	$<0, 2^{32} - 1>$	32-bitová celočíselná hodnota bez znamienka
long	System.Int64	$<-2^{63}, 2^{63} - 1>$	64-bitová celočíselná hodnota so znamienkom
ulong	System.UInt64	$<0, 2^{64} - 1>$	64-bitová celočíselná hodnota bez znamienka
float	System.Single	$<-3,4 \times 10^{38}, 3,4 \times 10^{38}>$	32-bitová reálna hodnota s jednoduchou presnosťou
double	System.Double	$<-1,79 \times 10^{308}, 1,79 \times 10^{308}>$	64-bitová reálna hodnota s dvojnásobnou presnosťou

Tab. 1.1: Vzťah primitívnych hodnotových typov jazyka C# 3.0 a ekvivalentných systémových dátových typov

Spoločný typový systém vytvára unifikovanú typovú abstrakciu, ktorá je zdieľaná všetkými .NET-kompatibilnými programovacími jazykmi. To je citeľná výhoda, pretože jednotný typový systém napomáha úspešne rozvinúť koncepciu skutočnej jazykovej interoperability. Na platforme Microsoft .NET Framework 3.5 je teda

možné, aby medzi sebou komunikovali aplikácie (resp. komponenty), ktoré boli napísané v rôznych programovacích jazykoch. Povedané inak, aplikácia jazyka C# 3.0 môže využívať dynamicky viazanú knižnicu (*.dll) pripravenú v jazyku Visual Basic 2008 bez toho, aby existovalo riziko vzniku programových chýb plynúcich z nekorektného použitia dátových typov, resp. typovej inkompatibility.



Obr. 1.5: Spoločný typový systém umožňuje spoluprácu aplikácií pripravených v rôznych .NET-kompatibilných programovacích jazykoch

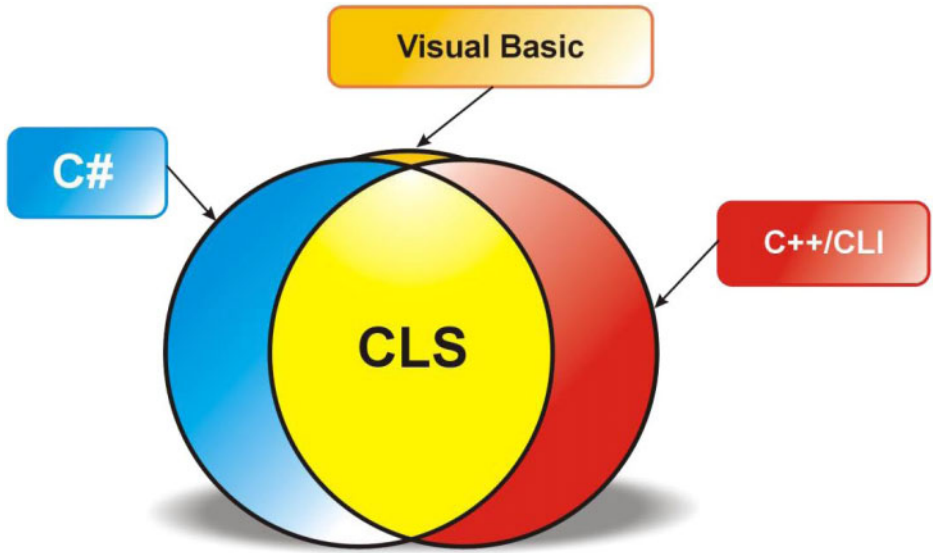
1.4 Spoločná jazyková špecifikácia (CLS)

Spoločný typový systém je zárukou toho, že aplikácie napísané v rôznych programovacích jazykoch vývojovo-exekučnej platformy Microsoft .NET Framework 3.5 môžu medzi sebou viesť plnohodnotný informačný dialóg. Lenže okrem dátových typov je nutné štandardizovať tiež schopnosti programovacích jazykov a ich kompilátorov. Kolekcia základných požiadaviek, ktoré musí spĺňať akýkoľvek .NET-kompatibilný programovací jazyk, formuje spoločnú jazykovú špecifikáciu. Spoločná jazyková špecifikácia napríklad vraví, ako môžu programátori používať primitívne dátové typy, či ako vytvárať nové, používateľsky deklarované dátové

typy. Spoločná jazyková špecifikácia tiež determinuje syntakticko-sémantické programové konštrukcie, ktoré súvisia s inštanciáciou tried, implementáciou rozhraní, dedičnosťou, polymorfizmom a aplikáciou ďalších elementov objektovo orientovanej paradigmy programovania. V neposlednom rade spoločná jazyková špecifikácia zaručuje spracovanie typovo bezpečných operácií na softvérových objektoch.

Z technického hľadiska vymedzuje spoločná jazyková špecifikácia minimálnu množinu syntakticko-sémantických konštrukcií, ktorými musí disponovať ľubovoľný .NET-kompatibilný programovací jazyk. To samozrejme neznamená, že daný programovací jazyk nemôže obsahovať aj iné, tzv. komplementárne programové konštrukcie. Tieto komplementárne konštrukcie však môžu pôsobiť len ako nadstavba primárnych konštrukcií, ktorých prítomnosť vyžaduje spoločná jazyková špecifikácia. Napríklad, navzdory tomu, že spoločná jazyková špecifikácia nedefinuje použitie smerníkov a smerníkovej aritmetiky, programovací jazyk C# 3.0 umožňuje vývojárom používať smerníky a realizovať ich aritmetiku v blokoch tzv. nebezpečného (unsafe) zdrojového kódu. No jazyk Visual Basic 2008 použitie smerníkov nepodporuje, takže spolupráca v tejto oblasti s jazykom C# 3.0 nie je možná. Je teda potrebné poukázať na skutočnosť, že zatiaľ čo syntakticko-sémantické programové konštrukcie definované spoločnou jazykovou špecifikáciou sú garantované naprieč všetkými .NET-kompatibilnými programovacími jazykmi, pre komplementárne programové konštrukcie to neplatí.

Koreláciu medzi syntakticko-sémantickými programovými konštrukciami implementovanými v jazykoch C# 3.0, C++/CLI a Visual Basic 2008 môžeme znázorniť pomocou Vennových diagramov (obr. 1.6).



Obr. 1.6: Spoločná jazyková špecifikácia determinuje množinu syntakticko-sémantických programových konštrukcií, ktoré musí obsahovať akýkoľvek .NET-kompatibilný programovací jazyk

2 Zostavenie riadenej aplikácie a jeho štruktúra

Zostavenie[5] je pomenovanie pre množinu spriaznených súborov, medzi ktorými existujú vzájomné prepojenia a ktoré sú pre potreby správy, riadenia verzií a distribúcie považované za samostatnú logickú jednotku. Ak vravíme o zostavení riadenej aplikácie, tak máme na mysli všetky súčasti, z ktorých je daná aplikácia zložená. Zostavenie je spravidla tvorené nasledujúcimi komponentmi:

- **Manifest.** Každé zostavenie riadenej aplikácie musí obsahovať manifest, ktorý charakterizuje zostavenie a vymedzuje jeho internú štruktúru. V manifeste sú zoskupené tieto informácie:

[5] Termín „zostavenie“ je v origináli označovaný termínom „assembly“.

- Názov zostavenia.
- Zoznam súborov, ktoré sú súčasťou zostavenia (spoločne s ich kryptografickými hešovými kódmi).
- Zoznam importovaných a exportovaných zostavení a dátových typov.
- Číselná identifikácia verzie zostavenia v tvare:

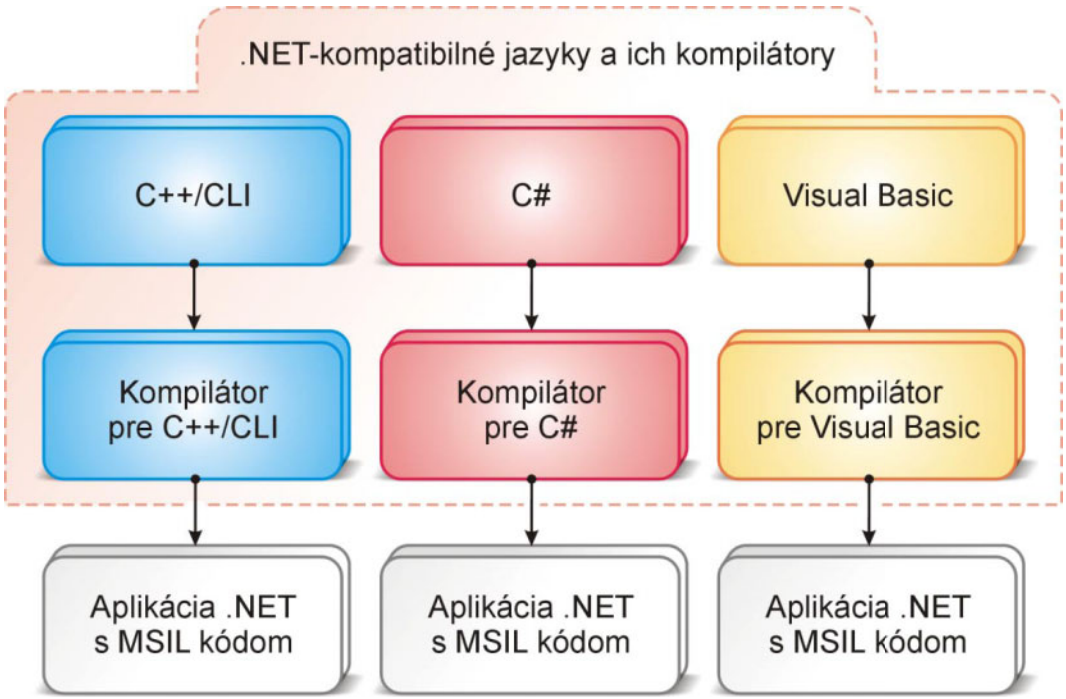
  H.V.Z.R

  - kde:
    - H je hlavné číslo verzie.
    - V je vedľajšie číslo verzie.
    - Z je číslo zostavenia.
    - R je číslo revízie.

- Textová identifikácia kultúry zostavenia.

Na každé zostavenie pripadá práve jeden manifest.

- **MSIL kód.** Zdrojový kód jazyka C# 3.0, podobne ako zdrojový kód akéhokoľvek iného .NET-kompatibilného programovacieho jazyka, je kompilátorom preložený do formy MSIL kódu. MSIL kód je nízkoúrovňový objektový pseudostrojový kód, pretože procesory triedy x86 nedokážu jeho inštrukcie explicitne spracovať. Naopak, MSIL kód je strojovým kódom pre virtuálny exekučný systém. Kód MSIL zavádza unifikáciu, pretože umožňuje prakticky realizovať ideu jazykovej interoperability na platforme .NET (obr. 2.1).



Obr. 2.1: Jazyková interoperabilita je dosiahnuteľná vďaka unifikovanému kódu pseudostrojového jazyka MSIL

To znamená, že aplikácie vytvorené v rôznych .NET-kompatibilných programovacích jazykoch môžu bez problémov spolupracovať. To isté platí aj pre komponenty: jedna aplikácia smie byť zložená z viacerých komponentov, pričom komponenty môžu byť vyvinuté v rôznych .NET-kompatibilných programovacích jazykoch.

- **Typové metadáta.** Typové metadáta sú dáta o dátových typoch, s ktorými riadená aplikácia pracuje. Vzhľadom na to, že informácie o všetkých používaných dátových typoch sú integrované v zostavení riadenej aplikácie, táto aplikácia nie je explicitne závislá od externých typových, resp. objektových knižníc. Eliminácia uvedenej závislosti je veľkou výhodou, najmä ak uvážime, aká ťažkopádna bola práca s typickými COM aplikáciami a ich typovými knižnicami. Keďže dáta o dátových typoch sú uskladnené

v zostavení riadenej aplikácie, môžeme s nimi kedykoľvek pracovať. Dynamickú inšpekciu typových metadát smieme realizovať pomocou mechanizmu reflexie. Reflexia má podobné vlastnosti ako dynamická identifikácia dátových typov (RTTI, Run-time Type Identification), ktorú poznáme z jazyka C++.

- **Natívne inštrukcie x86 na inicializáciu virtuálneho exekučného systému.** Operačné systémy triedy Windows sú natívnymi aplikáciami. Podobne aj virtuálny exekučný systém je natívnou aplikáciou (hoci v jeho područí bežia riadené aplikácie). Keďže je nutné pred štartom riadenej aplikácie korektne spustiť a inicializovať virtuálny exekučný systém, musia byť v zostavení riadenej aplikácie prítomné štartovacie bloky natívneho kódu. Ich úlohou je spustiť virtuálny exekučný systém a uviesť ho do okamžite použiteľného stavu. Keď používateľ (prípadne iný proces) zašle operačnému systému požiadavku na spustenie aplikácie .NET, operačný systém sa skutočnosť, že ide o riadenú aplikáciu, dozvie z príznakov, ktoré sú uložené v hlavičke priamo spustiteľného súboru riadenej aplikácie. Pretože operačný systém vie, že spustenie riadenej aplikácie si vyžaduje naštartovanie a inicializáciu virtuálneho exekučného systému, podrobí exekúcii natívne inštrukcie, ktoré zahajujú činnosť virtuálneho exekučného systému.

- **Aplikačné zdroje.** Riadená aplikácia môže obsahovať rôzne zdroje, ku ktorým patria obrázky, tabuľky reťazcov, ikony, súbory elektronickej dokumentácie, manuály či videosúbory interaktívnych kurzov. Všetky tieto zdroje môžu programátori zahrnúť do zostavenia riadenej aplikácie.

2.1 Klasifikácia zostavení riadených aplikácií

Nasledujúci zoznam uvádza klasifikáciu zostavení riadených aplikácií podľa rôznych kritérií:

1. kritérium: **Počet súborov**:

    - **Jednosúborové zostavenia.** Ak obsahuje zostavenie riadenej aplikácie len 1 súbor, tak ide o jednosúborové zostavenie. K tejto kategórii zostavení patria napríklad zostavenia štandardných konzolových aplikácií (ich obsahom je práve jeden priamo spustiteľný súbor).

    - **Viacsúborové zostavenia.** Ak obsahuje zostavenie riadenej aplikácie 2 a viac súborov, tak ide o viacsúborové zostavenie. Viacsúborovým zostavením je napríklad zostavenie, ktoré okrem priamo spustiteľného súboru obsahuje tiež dynamicky previazanú knižnicu so zdieľanou aplikačnou logikou.

2. kritérium**: Silné pomenovanie**:

    - **Zostavenia so slabým menom.** Ak nie je stanovené inak, všetky implicitne vytvorené zostavenia riadených aplikácií sú tzv. slabo pomenované zostavenia. Ak zostavenie disponuje slabým menom, nie je vybavené svojou identitou, ktorá sa skladá z týchto súčastí: meno, číslo verzie, kultúra, verejný kľúč a digitálny podpis.

    - **Zostavenia so silným menom.** Ak zostavenie obsahuje svoju identitu, ide o silno pomenované zostavenie. Zostavenia so silným menom smú okrem svojich dátových typov pracovať len s dátovými typmi iných silno pomenovaných zostavení.

3. kritérium: **Dynamické aspekty zostavenia**:

    - **Statické zostavenia.** Implicitne sú všetky vytvorené zostavenia riadených aplikácií statické. To znamená, že všetky súčasti zostavenia sú uložené na pevnom disku počítača, odkiaľ sa v prípade potreby načítavajú do operačnej pamäte počítača.

    - **Dynamické zostavenia.** Ak je zostavenie vytvorené a uložené v operačnej pamäti počítača, ide o dynamické zostavenie. Po svojej exekúcii môžu byť dynamické zostavenia uložené na pevný disk (tým sa z nich stanú statické zostavenia).

3 Technický rozbor riadenej exekúcie jednovláknovej riadenej aplikácie

Riadená exekúcia je termín, ktorým označujeme proces spustenia a nasledujúceho spracovania programových inštrukcií riadenej aplikácie. Adjektívum „riadená“ implikuje prítomnosť virtuálneho exekučného systému, ktorý riadi životné cykly riadených aplikácií.

Riadená exekúcia jednovláknovej riadenej aplikácie sa skladá z týchto etáp:

1. **Vytvorenie fyzického procesu.** Keď operačný systém zistí, že má byť spustená riadená aplikácia, pridelí jej súvislý adresovateľný pamäťový priestor. Tento pamäťový priestor je známy ako fyzický proces riadenej aplikácie. Na 32-bitových verziách operačného systému Windows je každej riadenej aplikácii pridelený fyzický proces so 4-gigabajtovým virtuálnym adresovateľným pamäťovým priestorom. Z pohľadu operačného systému je fyzický proces základným izolačným primitívom, ktoré dovoľuje súbežné spracovanie viacerých počítačových programov.

2. **Spustenie virtuálneho exekučného systému a jeho inicializácia.** Po alokácii fyzického procesu dochádza k aktivácii virtuálneho exekučného systému, ktorý bude zodpovedný za manažovanie životného cyklu riadenej aplikácie.

3. **Vytvorenie primárnej aplikačnej domény.** Podobne, ako sa operačný systém spolieha na fyzické procesy čoby izolačné primitíva, tak virtuálny exekučný systém nahliada na aplikačné domény. Aplikačná doména je logický proces, ktorý existuje v rámci fyzického procesu. V jednom fyzickom procese môže koexistovať viacero aplikačných domén. Navyše, v každej aplikačnej doméne smie bežať jedna riadená aplikácia, alebo jedna časť riadenej aplikácie. Spravovanie kontextu medzi viacerými aplikačnými doménami je oveľa efektívnejšie ako riadenie viacerých fyzických procesov. Každá riadená aplikácia má jednu hlavnú, tzv. primárnu aplikačnú doménu, ktorú automaticky vytvára virtuálny exekučný systém. Ďalšie, tzv. pracovné alebo tiež vedľajšie aplikačné domény, vytvára programátor podľa svojich potrieb.

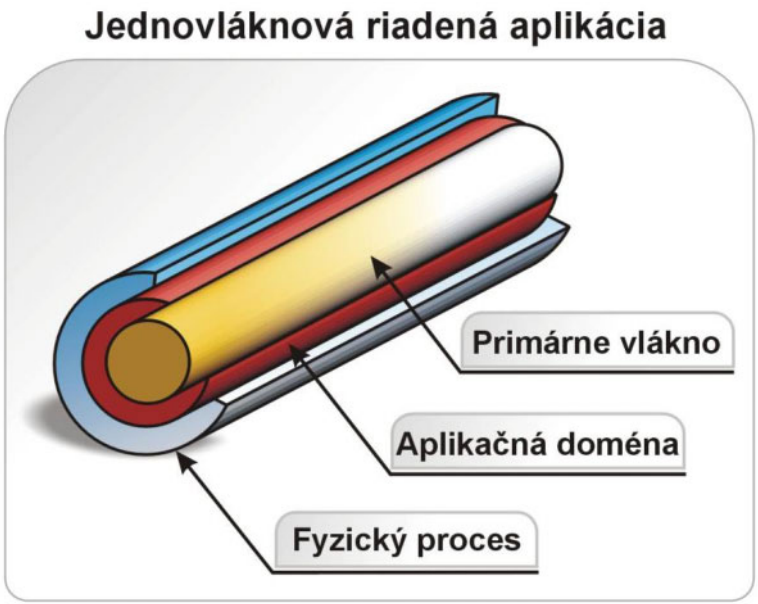
4. **Vytvorenie primárneho programového vlákna[6].** Programové vlákno predstavuje exekučný kanál, na ktorom sú spracúvané inštrukcie riadenej aplikácie. Virtuálny exekučný systém vkladá do primárnej aplikačnej domény práve jedno primárne programové vlákno, na ktorom sa bude uskutočňovať exekúcia programových príkazov. Nech n je počet programových vlákien riadenej aplikácie. Potom môžeme zaviesť nasledujúcu kategorizáciu:

   - Ak $n = 1 \Rightarrow$ ide o jednovláknovú riadenú aplikáciu.
   - Ak $n \geq 2 \Rightarrow$ ide o viacvláknovú riadenú aplikáciu.

   Implicitne sú všetky riadené aplikácie jednovláknové. Primárne programové vlákno však môže v prípade potreby vytvárať aj ďalšie, tzv. pracovné, resp. vedľajšie programové vlákna. V tejto kapitole sa budeme venovať

[6] Termín „vlákno“ sa niekedy stotožňuje s termínom „podproces“.

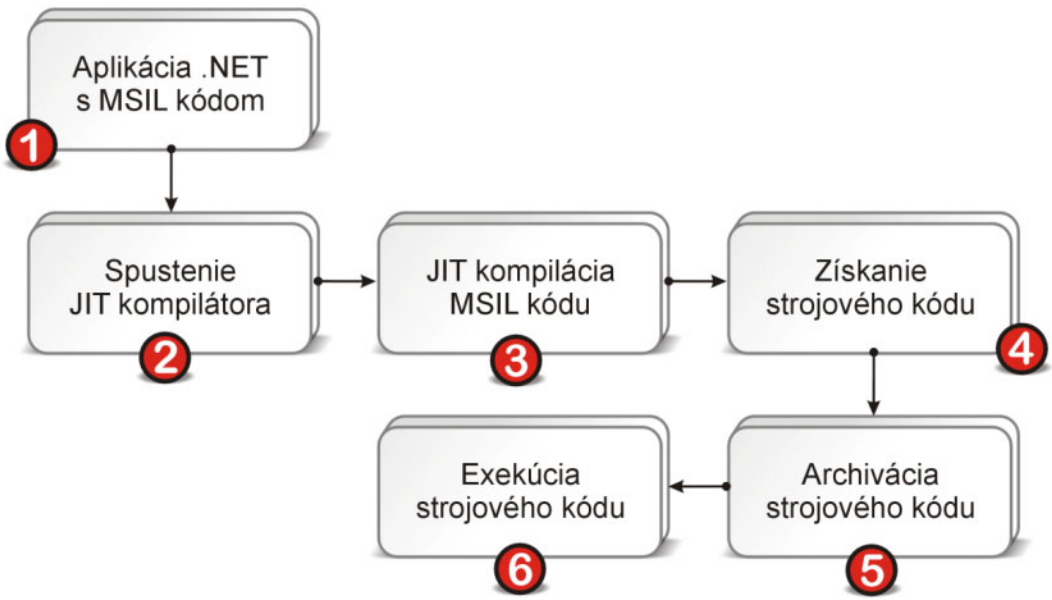
ozrejmeniu podstaty riadenej exekúcie jednovláknovej riadenej aplikácie. Riadenú exekúciu viacvláknovej riadenej aplikácie rozoberieme v kapitole *4 Technický rozbor riadenej exekúcie viacvláknovej riadenej aplikácie*. Vizuálny model jednovláknovej riadenej aplikácie je zobrazený na obr. 3.1.



Obr. 3.1: Jednovláknová riadená aplikácia

5. **Just-In-Time (JIT) kompilácia MSIL kódu.** Len čo je primárne programové vlákno pripravené, dochádza k prekladu MSIL kódu do podoby strojového kódu pomocou Just-In-Time (JIT) kompilátora a k exekúcii získaného strojového kódu na primárnom programovom vlákne. MSIL kód, nachádzajúci sa v zostavení riadenej aplikácie, bude v konečnom dôsledku preložený do strojového kódu triedy x86. Za bežných okolností realizuje preklad MSIL kódu do strojového kódu JIT kompilátor. JIT kompilátor je stroj, ktorý je súčasťou virtuálneho exekučného systému. Ak MSIL kód prekladá JIT kompilátor, ide o proces, ktorý nazývame preklad na požiadanie. JIT kompilátor je spravidla povolaný vždy, ak neexistuje vopred skompilovaná natívna verzia riadenej aplikácie. JIT kompilátor prekladá

jednotlivé metódy MSIL kódu podľa priority ich exekúcie, pričom raz preložené metódy sú uskladnené do vyhradenej pamäťovej oblasti primárnej aplikačnej domény riadenej aplikácie. Ak je metóda raz preložená a príde požiadavka na jej opätovnú aktiváciu, JIT kompilátor nebude MSIL kód príslušnej metódy kompilovať znova, ale použije už preloženú (natívnu) verziu metódy[7].

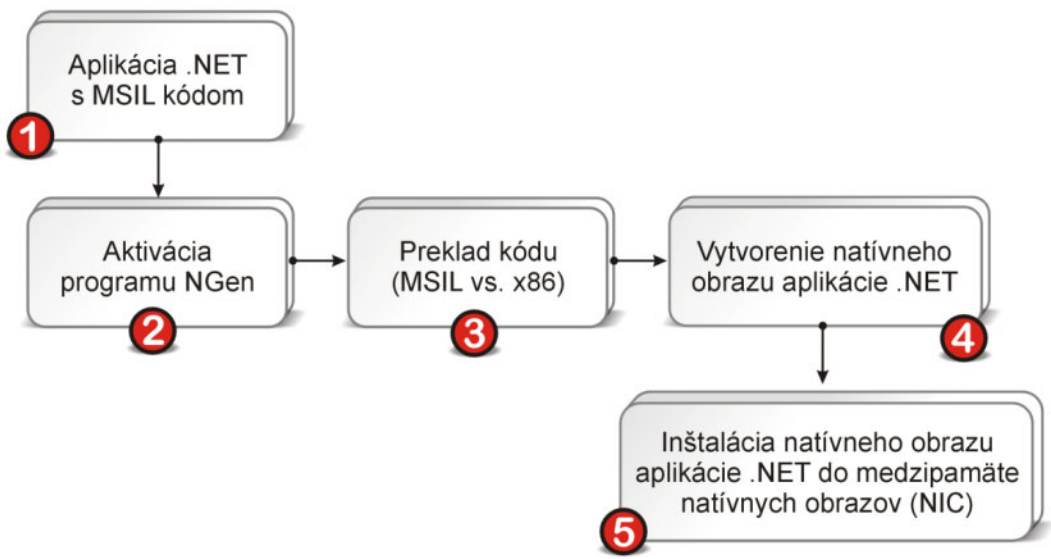


Obr. 3.2: JIT kompilácia MSIL kódu

Výhodou JIT kompilácie je schopnosť JIT kompilátora optimalizovať výsledný strojový kód vzhľadom na inštrukčnú súpravu procesora okamžite počas prekladu na požiadanie. Slabou stránkou JIT kompilácie je istá

[7] Strojový kód preložených metód je štandardne vždy ukladaný za účelom jeho efektívneho budúceho použitia. Avšak môže sa stať, že virtuálny exekučný systém bude nútený strojový kód už preložených metód zlikvidovať v záujme uvoľnenia čo možno najväčšieho pamäťového priestoru. K tejto situácii dochádza pri kritickom využití systémových prostriedkov, kedy virtuálny exekučný systém preferuje maximalizáciu alokovateľného pamäťového priestoru pred optimalizáciou exekučného času JIT kompilácie.

postranná réžia, ktorá sa viaže najmä s inicializáciou JIT kompilátora. Táto postranná réžia môže zapríčiniť určitú latenciu pri spúšťaní riadenej aplikácie, čo môže v konečnom dôsledku nepriaznivo ovplyvniť subjektívnu spokojnosť používateľa so štartom riadenej aplikácie. Pri vývoji kritických aplikácií, ktoré nepripúšťajú žiadne štartovacie latencie, sa preto odporúča MSIL kód riadenej aplikácie vopred pretransformovať do strojového kódu. Tento, tzv. počiatočný natívny preklad, sa uskutočňuje obvykle v poslednom inštalačnom štádiu riadenej aplikácie. Výsledkom po uskutočnení počiatočného natívneho prekladu, je natívne zostavenie pôvodne riadenej aplikácie. Toto natívne zostavenie obsahuje natívny obraz (teda rýdzi strojový kód) a je inštalované do medzipamäte natívnych obrazov (NIC, Native Image Cache). Natívne zostavenia môžeme z riadených zostavení zhotoviť programom Native Image Generator (skrátene NGen). Konštrukciu natívneho zostavenia znázorňuje obr. 3.3.



Obr. 3.3: Tvorba natívnych obrazov riadených aplikácií

JIT kompilácia sa uskutočňuje dovtedy, kým:

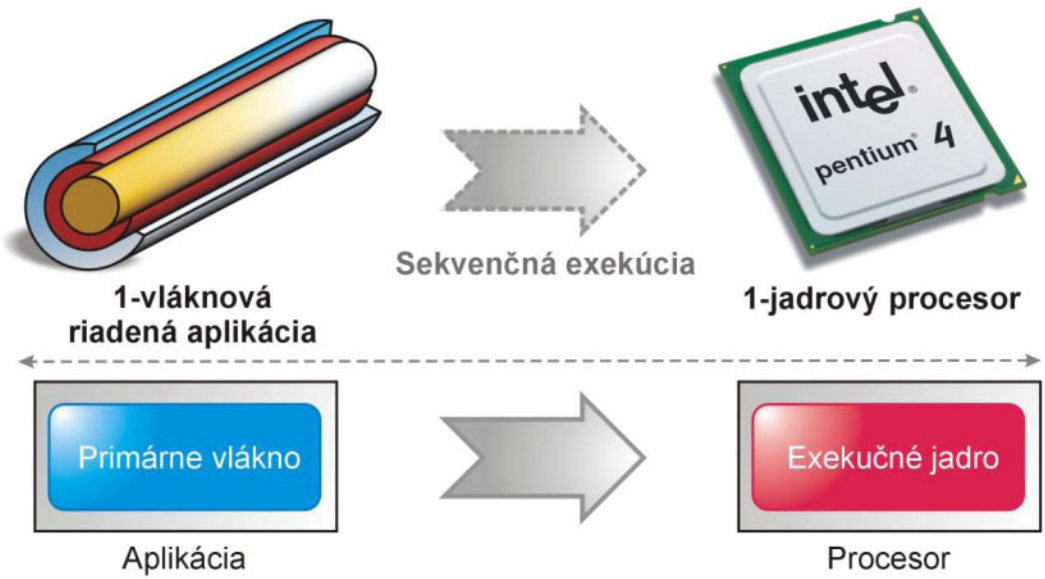
- existujú nepreložené metódy (v MSIL kóde),
- používateľ, alebo iný proces, neukončí beh riadenej aplikácie.

Životný cyklus riadenej aplikácie sa končí v momente, keď je doručená požiadavka na ukončenie aplikácie. V tejto chvíli sa najskôr ukončí spracovanie programových inštrukcií na primárnom programovom vlákne. Následne virtuálny exekučný systém zlikviduje primárne programové vlákno, pričom deštrukcii podrobí tiež primárnu aplikačnú doménu. Proces pokračuje terminációu virtuálneho exekučného systému a dealokáciou fyzického procesu (túto činnosť uskutočňuje operačný systém). Tak dochádza k uvoľneniu všetkých systémových prostriedkov, ktorých pridelenie si jednovláknová riadená aplikácia vyžadovala. Keďže zdroje sú obnovené, môžu byť použité pri ďalších aktivitách, ktoré operačný systém vykonáva.

Riadená exekúcia jednovláknovej riadenej aplikácie je sekvenčná. To znamená, že jednotlivé inštrukcie sú na primárnom programovom vlákne spracúvané sekvenčne, teda v presnej lineárnej postupnosti (jedna inštrukcia za druhou). Sekvenčné spracovanie inštrukcií môžeme charakterizovať synchrónnym exekučným modelom. Synchrónny exekučný model neumožňuje paralelizáciu činností. V záujme detailnejšej technickej analýzy smieme skúmať exekúciu jednovláknovej riadenej aplikácie na jednojadrovom procesore a na viacjadrovom procesore. Závery, ku ktorým dospejeme, budú analogicky platiť aj pre exekúciu jednovláknovej riadenej aplikácie na jednoprocesorových a viacprocesorových strojoch.

1. **model: Exekúcia jednovláknovej riadenej aplikácie na jednojadrovom procesore.** Toto je sekvenčná exekúcia, ktorú môžeme opísať sekvenčným exekučným modelom. Jednojadrový procesor počítača dokáže v jednom časovom momente spracúvať tok inštrukcií práve jedného programového vlákna. V tejto súvislosti by sme radi poznamenali, že v jednom hodinovom cykle procesora môžu byť spracované viaceré mikroinštrukcie súčasne. Ak dokáže procesor vykonať viacero mikroinštrukcií súčasne v jednom pracovnom cykle, ide o tzv. superskalárny procesor. Inštrukcie strojového

kódu, ktoré sú finálnym produktom JIT kompilátora, sú ďalej segmentované na mikroinštrukcie, ktoré sú posielané priamo procesoru. Pre potreby efektívnej exekúcie mikroinštrukcií môže procesor zmeniť poradie ich spracúvania, samozrejme so zachovaním ich pôvodnej funkcionality (táto technika sa nazýva Out-of-Order Execution, teda spracovanie mikroinštrukcií v inom poradí, než v akom boli pôvodne procesoru poskytnuté). Hoci je procesor schopný do určitej miery optimalizovať exekúciu mikroinštrukcií jednovláknovej riadenej aplikácie, ide o rýdzo sekvenčnú exekúciu, ktorá nepracuje s paralelizmom.

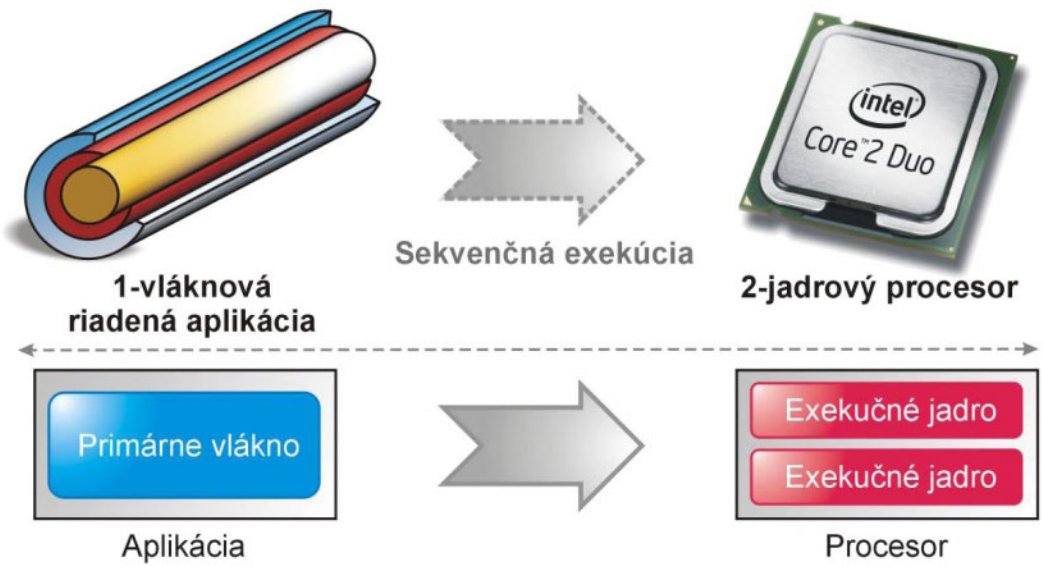


Obr. 3.4: Sekvenčná exekúcia jednovláknovej riadenej aplikácie na jednojadrovom procesore

2. **model: Exekúcia jednovláknovej riadenej aplikácie na viacjadrovom procesore.** Ak rozšírime výpočtovú kapacitu systému dodaním viacerých exekučných jadier procesora, tak beh jednovláknovej riadenej aplikácie bude opäť sekvenčný. Je to preto, že spracúvaním toku inštrukcií primárneho programového vlákna bude zaneprázdnené len jedno exekučné jadro procesora. Ostatné exekučné jadrá nebudú využité, pretože neexistujú

žiadne ďalšie programové vlákna, ktoré by mohli byť na zvyšných exekučných jadrách procesora realizované. Z uvedeného vyplýva, že spustenie jednovláknovej riadenej aplikácie na viacjadrovom procesore neprináša žiaden implicitný nárast jej výkonnosti.

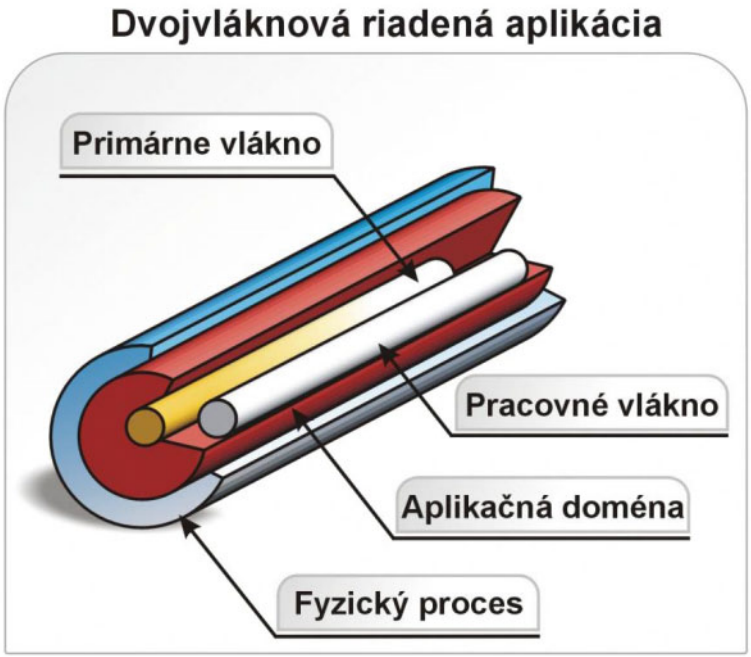
3. Hoci jedno exekučné jadro bude pracovať na maximum, celkové využitie výpočtovej kapacity systému nikdy nepresiahne hodnotu $\left(\frac{1}{n}\right) * 100$ [%], kde n je počet exekučných jadier viacjadrového procesora. V prípade 4-jadrového procesora bude platiť toto: $\left(\frac{1}{4}\right) * 100 = 25$ %. Spustenie jednovláknovej riadenej aplikácie na počítači osadenom 4-jadrovým procesorom znamená len 25-percentné využitie výkonnostného potenciálu stroja.



Obr. 3.5: Sekvenčná exekúcia jednovláknovej riadenej aplikácie na viacjadrovom procesore

4 Technický rozbor riadenej exekúcie viacvláknovej riadenej aplikácie

Ak disponuje riadená aplikácia aspoň dvomi programovými vláknami, označujeme ju ako viacvláknovú. Jedno z vlákien i naďalej zostává primárnym programovým vláknom, zatiaľ čo ostatné vlákna vystupujú ako pracovné vlákna. Keď vyvíjame viacvláknovú aplikáciu, mali by sme vždy presne vymedziť pracovné modely jednotlivých vlákien.



Obr. 4.1: Viacvláknová riadená aplikácia

Viacvláknová aplikácia umožňuje pseudoparalelnú, alebo rýdzo paralelnú exekúciu tokov programových inštrukcií, v závislosti od toho, na akej počítačovej stanici je spustená. Podobne ako pri analýze riadenej exekúcie jednovláknovej riadenej aplikácie, tak aj pri skúmaní spracovania viacvláknovej riadenej aplikácie stanovíme dva základné exekučné modely:

1. **model: Exekúcia viacvláknovej riadenej aplikácie na jednojadrovom procesore.** Tento exekučný model označujeme termínom pseudoparalelná exekúcia. Vzhľadom na to, že jednojadrový procesor má iba jedno exekučné jadro, v jednom okamihu smie spracúvať programové inštrukcie práve jedného programového vlákna. Ak je riadená aplikácia viacvláknová, v jednej aplikačnej doméne bude koexistovať množina programových vlákien. Operačné systémy triedy Windows (ale samozrejme aj iné moderné operačné systémy) podporujú viacvláknové spracovanie riadených aj natívnych aplikácií. Vďaka preemptívnemu viacvláknovému spracovaniu sú viacvláknové aplikácie spracúvané nasledujúcim iteratívnym spôsobom:

    - Každému programovému vláknu je pridelené tzv. časové kvantum. Časové kvantum je najmenšia jednotka alokácie výpočtovej kapacity procesora. Ak vlákno získa časové kvantum, všetky výpočtové zdroje jednojadrového procesora sa budú sústrediť na vykonanie programových inštrukcií daného vlákna. Pritom platí, že ako prvému bude časové kvantum poskytnuté primárnemu programovému vláknu (to je pochopiteľné, keďže primárne vlákno je asociované s metódou, ktorá predstavuje vstupný bod viacvláknovej aplikácie).

    - Vo chvíli, keď časové kvantum primárneho programového vlákna pominie, procesor v súčinnosti s operačným systémom zabezpečí uchovanie aktuálneho stavu spracovania tohto programového vlákna. Stav spracovania musí byť vždy archivovaný, pretože predstavuje významný ukazovateľ pracovného progresu vlákna, ktorý bude využitý pri nasledujúcej interakcii s vláknom.

    - Procesor sa v ďalšom kroku presúva na nasledujúce pracovné programové vlákno a začína spracúvať jeho tok programových inštrukcií. Výpočtová kapacita procesora je dedikovaná tomuto vláknu, kým neuplynie jeho časové kvantum. Ak sa tak stane, uloží sa aktuálny stav spracovania tohto vlákna a procesor sa presúva na ďalšie vlákno v poradí. Poradie, v akom sú jednotlivé vlákna

spracúvané, je ovplyvnené ich prioritou. Každé vlákno smie disponovať jedným z niekoľkých prioritných stupňov. Hoci z pohľadu operačného systému existuje značne široká škála prioritných stupňov programových vlákien, vývojári pracujúci v jazyku C# 3.0 majú na výber z piatich prioritných stupňov. Interakcia s programovými vláknami podľa ich priority znamená, že najskôr budú realizované vlákna s vyššími prioritnými stupňami, a potom vlákna s nižšou prioritou[8].

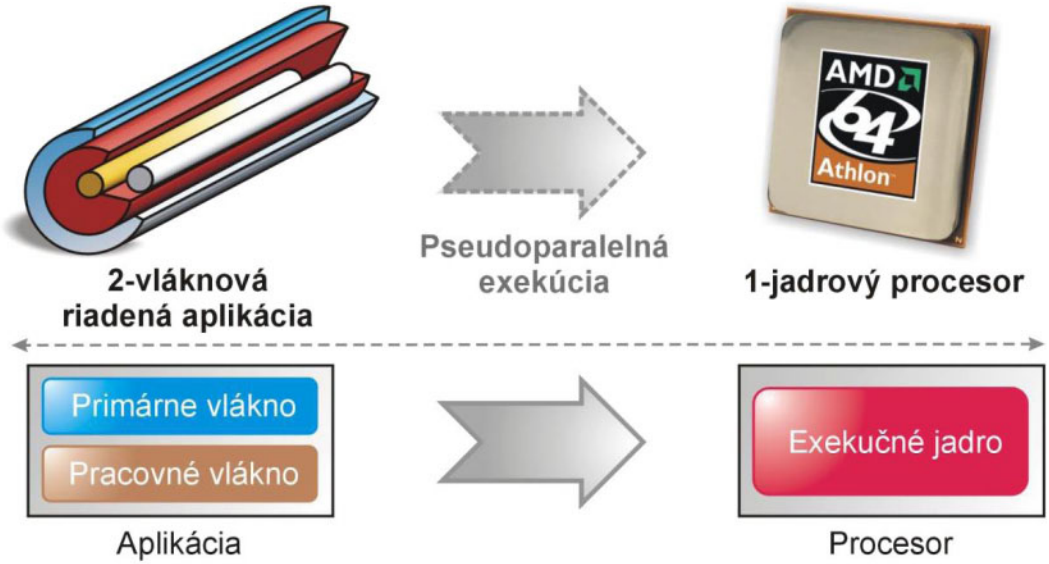
- Naznačená migrácia medzi vláknami sa odohráva až dovtedy, kým procesor neobslúži aj posledné programové vlákno. Tým sa dokončí jedna iterácia, v rámci ktorej boli realizované programové inštrukcie všetkých vlákien tvoriacich viacvláknovú riadenú aplikáciu. Keďže fyzikálna dĺžka časového kvanta je krátka[9], z vonkajšieho pohľadu sa zdá, že programové vlákna pracujú súbežne, a teda paralelne. Nie je to však tak, lebo paralelné spracovanie by znamenalo vykonanie tokov inštrukcií viacerých vlákien naraz, čiže v rovnakom čase. Iteratívne spracovanie viacvláknovej aplikácie na jednojadrovom procesore však tejto požiadavke nezodpovedá, a preto ho označujeme termínom pseudoparalelné.

- Len čo je hotová jedna iterácia pseudoparalelného spracovania programových vlákien viacvláknovej aplikácie, začína sa ďalšia iterácia. V nej procesor v súčinnosti s operačným systémom znovu spracúva inštrukcie jednotlivých vlákien, avšak samozrejme pokračuje tam, kde v predchádzajúcej iterácii skončil (bod pokračovania je jednoznačne determinovaný archivovaným stavom spracovania programového vlákna). Iterácie v naznačenom poradí sa

[8] Mohlo by sa zdať, že vlákna s malou prioritou nebudú obslúžené tak často ako je potrebné, resp. že za určitých okolností nebudú obslúžené vôbec. Nie je to tak. Operačný systém v spolupráci s virtuálnym exekučným systémom monitoruje priebeh spracovania programových vlákien a pokiaľ by sa stalo, že niektoré vlákna sú dlhší čas nečinné, môže im byť automaticky zvýšená priorita tak, aby dokázali získať časové kvantum.

[9] Veľkosť časového kvanta je variabilná, no rádovo sa pohybuje v milisekundových intervaloch.

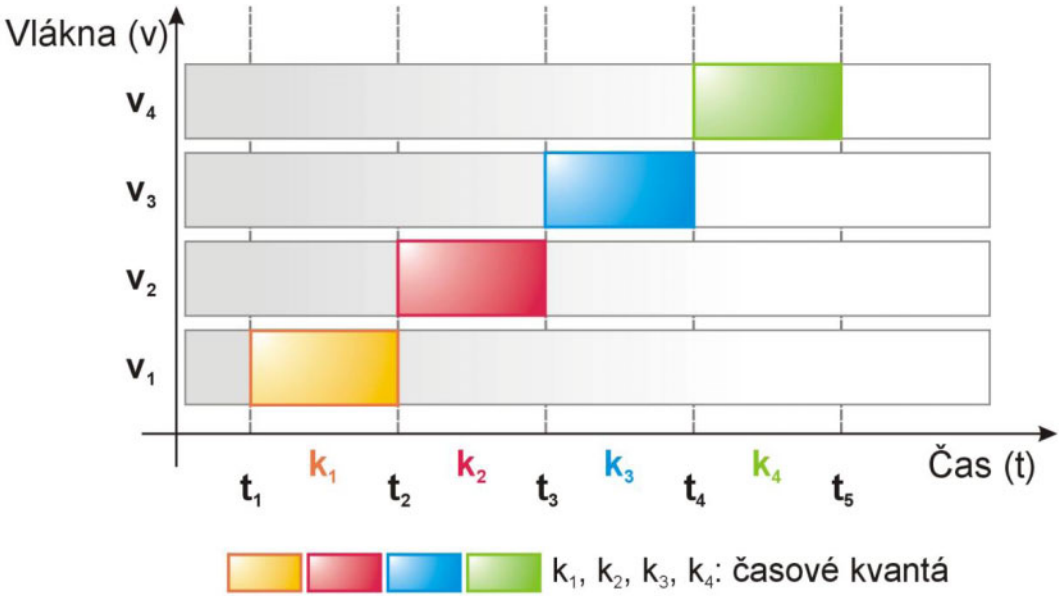
opakujú až do konca životného cyklu viacvláknovej riadenej aplikácie[10].



Obr. 4.2: Pseudoparalelná exekúcia viacvláknovej riadenej aplikácie na jednojadrovom procesore

Proces zdieľania výpočtových zdrojov jednojadrového procesora pri riadenej exekúcii viacvláknovej riadenej aplikácie pomocou prideľovania časových kvánt je zobrazený na obr. 4.3.

[10] Samozrejme, môže sa stať, že niektoré pracovné programové vlákno zanikne. Za týchto okolností ďalšie iterácie pracujú len s tými vláknami, ktoré sú v pohotovostnom stave (ide o tzv. živé vlákna).



Obr. 4.3: Alokácia časových kvánt pri pseudoparalelnej exekúcii

Komentár k obr. 4.3: Schéma zobrazuje proces alokácie časových kvánt jednotlivým vláknam viacvláknovej riadenej aplikácie pri ich exekúcii na jednojadrovom procesore. V čase t_1 získa časové kvantum k_1 primárne programové vlákno $\mathbf{v_1}$. Počas plynutia časového kvanta sú výpočtové kapacity jednojadrového procesora k dispozícii výhradne primárnemu programovému vláknu. Keď časové kvantum k_1 v čase t_2 vyprší, procesor poskytuje časové kvantum k_2 prvému pracovnému vláknu ($\mathbf{v_2}$). Po spracovaní vlákna $\mathbf{v_2}$ a uplynutí časového kvanta k_2 je časové kvantum poskytnuté ďalšiemu pracovnému programovému vláknu v poradí ($\mathbf{v_3}$). Prideľovanie časových kvánt sa opakuje až pokiaľ nie sú obslúžené všetky programové vlákna riadenej aplikácie. To sa v našom prípade deje až do momentu, kedy uplynie časové kvantum k_4. Keďže naša aplikácia je 4-vláknová, v čase t_5 bude pridelené časové kvantum opäť prvému, a teda primárnemu vláknu aplikácie. Analogicky, v čase t_5+k_1 získa časové kvantum znova prvé pracovné vlákno ($\mathbf{v_2}$), v čase $t_5+k_1+k_2$ druhé pracovné vlákno ($\mathbf{v_3}$), atď.

V našom schematickom modeli definujeme časové kvantum takto: $k_i = t_{i+1} - t_i$, kde $i \in < 1, n >$ a $n \in N$. Zavádzame teda abstrakciu, že časové kvantá sú presne rovnaké. V praktických podmienkach sa však dĺžka časových kvánt môže do istej miery variabilne odlišovať, čiže presnejšie by sme mohli povedať, že časové kvantá poskytované programovým vláknam sú približne rovnaké. Z pohľadu našej analýzy si však môžeme dovoliť vplyv uvedených diferencií zanedbať.

2. **model: Exekúcia viacvláknovej riadenej aplikácie na viacjadrovom procesore.** Toto je variant rýdzo paralelnej exekúcie viacvláknovej riadenej aplikácie. Paralelná exekúcia znamená, že jednotlivé programové vlákna riadenej aplikácie budú presne namapované na exekučné jadrá procesora. V záujme dosiahnutia paralelnej exekúcie viacvláknovej aplikácie, je nutné prijať tieto predpoklady:

    1. **Optimálny počet programových vlákien.** Ideálne je, ak je počet programových vlákien rovný počtu exekučných jadier viacjadrového procesora. Ak teda pracujeme na stroji s 2-jadrovým procesorom, riadená aplikácia by mala obsahovať 2 programové vlákna, ktoré budú vykonávané paralelne. Analogicky, na 4-jadrovom procesore dospejeme k optimálnej kompozícii so 4 programovými vláknami. Vo všeobecnosti smieme vyhlásiť, že počet programových vlákien je lineárne závislý od počtu exekučných jadier procesora. Je preto nutné vyvarovať sa príliš nízkemu počtu programových vlákien (z dôvodu nevyužitia celkovej výkonnostnej kapacity počítačového systému) ako aj príliš vysokému počtu programových vlákien (nie je možná ich paralelná exekúcia z dôvodu nedostatočného počtu exekučných jadier procesora).

    2. **Škálovateľnosť programových vlákien.** Je dôležité poukázať na skutočnosť, že optimálny počet vlákien je na rôzne výkonných strojoch variabilný. V každom prípade je potrebné prijať premisu škálovateľného počtu programových vlákien. Škálovateľnosť

definujeme ako schopnosť viacvláknovej riadenej aplikácie flexibilne sa prispôsobiť hocako výkonnej hardvérovej platforme. Aj keď je pochopiteľné, že na menej výkonných strojoch bude viacvláknová riadená aplikácia podávať nižší výkon ako na vysokovýkonných pracovných staniciach, vždy musí operovať s optimálnym počtom vlákien pre aktuálnu hardvérovú infraštruktúru.

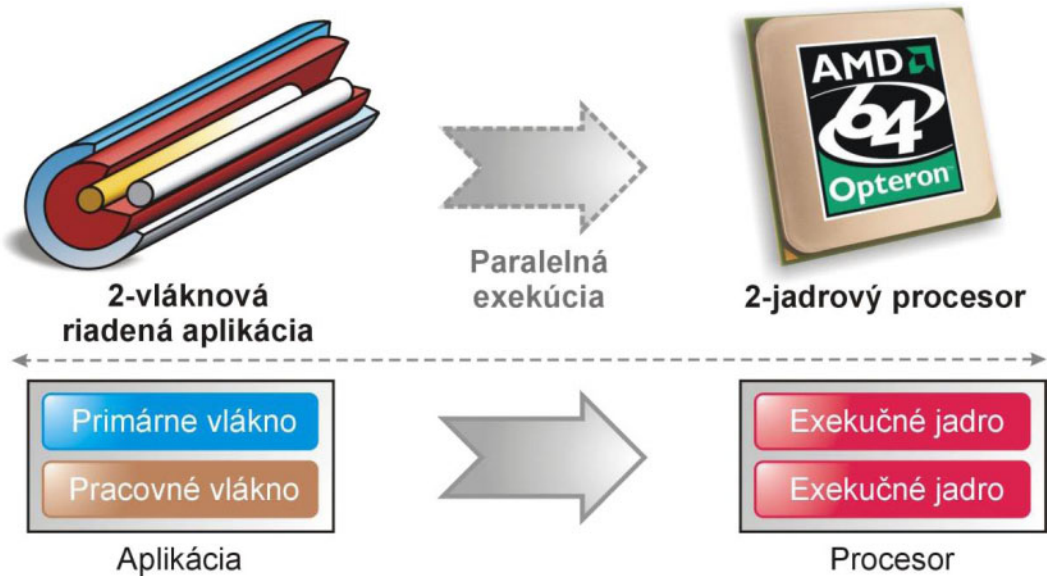
3. **Optimálna distribúcia pracovného zaťaženia programových vláken.** Úloha, ktorú viacvláknová riadená aplikácia rieši, musí byť algoritmizovaná tak, aby bolo zabezpečené rovnomerné rozloženie pracovného zaťaženia na všetky programové vlákna. V procese problémovej dekompozície je preto nutné exaktne determinovať činnosti, ktoré budú programové vlákna uskutočňovať, resp. bloky dátových štruktúr, s ktorými budú vlákna operovať. Ak je aplikácia zložená z n programových vlákien, každé z nich by malo riešiť $1/n$ celej úlohy. Ak je pracovné zaťaženie distribuované rovnomerne, dochádza k eliminácii stavov, v ktorých určité vlákno, resp. vlákna vykonajú svoju činnosť skôr ako ostatné vlákna. V danom kontexte by sme zaznamenali stav, kedy by isté vlákna boli zaneprázdnené vykonávaním svojich činností, zatiaľ čo iné vlákna by boli nevyužité.

4. **Eliminácia závislostí medzi viacerými programovými vláknami.** Keďže sa chceme vyhnúť kolíznym stavom, ktoré vznikajú predovšetkým pri konkurenčnom prístupe k zdieľaným objektom či dátovým zdrojom, bude nutné projektovať prácu programových vlákien tak, aby medzi nimi nevznikali žiadne nepriaznivé interferencie. Praktické riešenia týchto problémov sú vedené dvomi smermi:

1. **Vylúčenie zdieľaných zdrojov.** Z pôvodne zdieľaného zdroja môžeme urobiť nezdieľaný zdroj, a to tým spôsobom, že každému programovému vláknu, ktoré s príslušným zdrojom musí pracovať, poskytneme samostatnú kópiu tohto zdroja (táto technika za označuje termínom privatizácia zdrojov). Tým sa zdieľaný zdroj stane súkromným zdrojom programového vlákna.

2. **Zabezpečenie synchronizovaného prístupu k zdieľaným zdrojom.** Operačný systém, virtuálny exekučný systém, bázová knižnica tried a jazyková špecifikácia C# 3.0 umožňujú vývojárom používať mnohé synchronizačné primitíva, ktoré nariaďujú jednoznačný prístup k zdieľanému zdroju. Napriek tomu, že použitím synchronizačných primitív dokážeme predísť potenciálne nekonzistentnému stavu zdroja pri neriadenom prístupe, vždy sa z pôvodne paralelnej exekúcie dostávame k sekvenčnej exekúcii. Je to preto, že programové vlákno s garantovaným prístupom k zdieľanému zdroju má prednosť pred ostatnými vláknami (ktoré čakajú, kým aktuálne vlákno neumožní aj im pracovať so zdieľaným zdrojom).

Ak budeme predpokladať splnenie charakterizovaných predpokladov, môžeme konštatovať, že paralelná exekúcia viacvláknovej riadenej aplikácie na viacjadrovom procesore je najlepším exekučným modelom, najmä čo sa týka využitia celkového výkonnostného potenciálu počítača s viacjadrovým procesorom. Paralelizácia znamená súbežné vykonávanie rôznych úloh viacerými programovými vláknami, alebo súbežné uskutočňovanie operácií s rôznymi blokmi dátových štruktúr.



Obr. 4.4: Paralelná exekúcia viacvláknovej riadenej aplikácie
na viacjadrovom procesore

Špeciálnu pozornosť si vyžaduje riadená exekúcia viacvláknovej riadenej aplikácie, ktorej počet programových vlákien presahuje počet exekučných jadier procesora. Navzdory tomu, že nejde o ideálny stav, v praxi nie je výskyt aplikácií s uvedeným vláknovým zložením ojedinelý. V tomto kontexte smieme zaviesť špeciálny exekučný model, v ktorom budeme skúmať vykonanie viacvláknovej aplikácie na viacjadrovom procesore, pričom $n > E_X$, kde n je počet programových vlákien aplikácie a E_X je počet exekučných jadier procesora.

Špeciálny exekučný model: Exekúcia viacvláknovej riadenej aplikácie na viacjadrovom procesore, pričom $n > E_X$. V tomto exekučnom modeli dochádza ku kombinácii paralelnej a pseudoparalelnej exekúcie riadenej aplikácie. V ďalšom výklade budeme vychádzať z týchto premís:

1. Viacvláknová aplikácia obsahuje párny počet programových vlákien.
2. Viacjadrový procesor obsahuje párny počet exekučných jadier.

Virtuálny exekučný systém v kooperácii s operačným systémom zabezpečia vytvorenie súprav vlákien (S_V) a ich distribúciu na exekučné jadrá procesora. V ďalšej analýze budeme vychádzať z týchto predpokladov:

1. Počet súprav vlákien je zhodný s počtom exekučných jadier procesora, teda $S_V = E_X$.

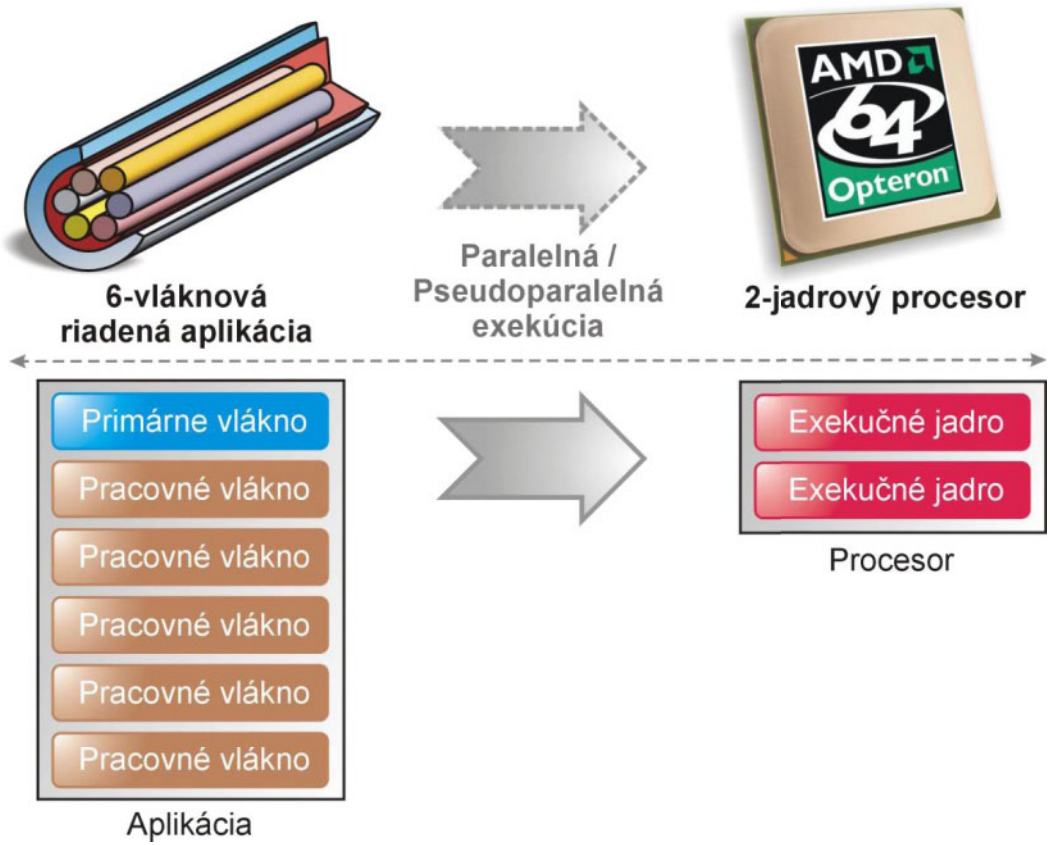
2. Absolútna početnosť (A_P) programových vlákien (P_V) je v jednotlivých súpravách vlákien identická, teda $A_P(P_V) = k$, kde k je konštanta.

Rozloženie súprav vlákien na exekučné jadrá procesora determinujeme funkciou, ktorá zabezpečuje generovanie jednoznačného vzťahu medzi súpravami vlákien a exekučnými jadrami procesora:

$$f: S_V \rightarrow E_X$$

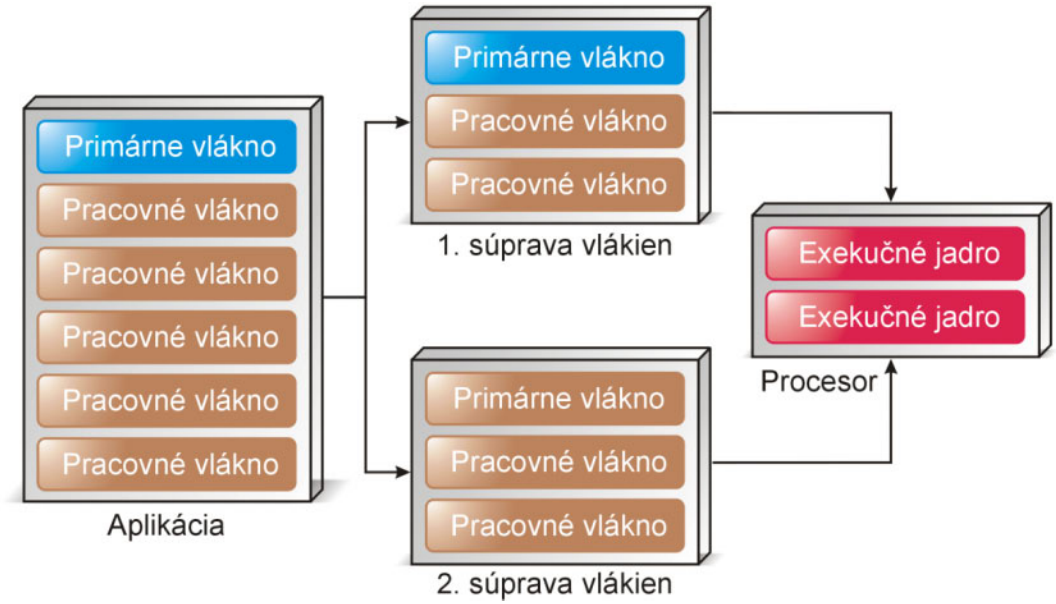
Keďže súprav vlákien je presne toľko, koľko je exekučných jadier procesora, môžeme konštatovať, že riadená exekúcia súprav vlákien je paralelná. Je to preto, že v jednom časovom okamihu sú súbežne spracúvané viaceré súpravy vlákien. Na druhej strane, každá súprava vlákien zahŕňa určitý počet programových vlákien. Ak na jedno exekučné jadro pripadá viacero programových vlákien, ich exekúcia nemôže byť paralelná. Na úrovni spracovania programových vlákien preto identifikujeme pseudoparalelnú exekúciu, ktorá je realizovateľná prostredníctvom zdieľania výpočtových kapacít exekučného jadra procesora. Zdieľanie exekučného jadra sa uskutočňuje pomocou mechanizmu časových kvánt, čiže časových dávok, ktoré sú vláknam prideľované podľa ich priority.

Na obr. 4.5 je znázornená paralelná/pseudoparalelná exekúcia viacvláknovej riadenej aplikácie, ktorej počet programových vlákien prevyšuje počet exekučných jadier procesora.



Obr. 4.5: Paralelná/pseudoparalelná exekúcia viacvláknovej riadenej aplikácie na viacjadrovom procesore

Na obr. 4.6 je zobrazený proces distribúcie programových vlákien viacvláknovej riadenej aplikácie do homogénnych súprav vlákien a ich následnej paralelnej exekúcie na jednotlivých exekučných jadrách viacjadrového procesora.



Obr. 4.6: Distribúcia programových vlákien do súprav a ich exekúcia

5 Typológia vlákien

V rámci komplexnej analýzy riadenej exekúcie jedno- a viacvláknových riadených aplikácií je potrebné dodať, že s vláknami sa stretávame na nasledujúcich troch úrovniach:

1. Programové vlákna.
2. Vlákna operačného systému.
3. Hardvérové vlákna procesora.

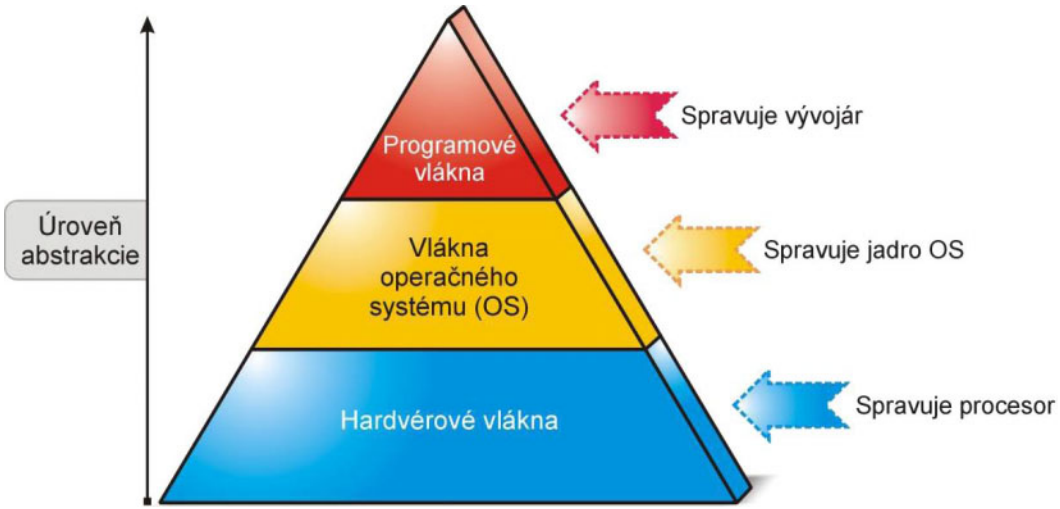
Doposiaľ sme explicitne pracovali len s 1. úrovňou vlákien, a teda vláknami programovými. Programové vlákna sú vlákna, s ktorými pracujeme pomocou syntakticko-sémantických konštrukcií programovacieho jazyka C# 3.0, resp. pomocou prostriedkov bázovej knižnice tried vývojovo-exekučnej platformy Microsoft .NET Framework 3.5. Práca s programovými vláknami sa vyznačuje najvyššou mierou abstrakcie, pretože v tomto ponímaní sú programové vlákna inštanciami triedy **Thread** z menného priestoru **System.Threading**. Bližšie informácie o explicitnej konštrukcii a používaní programových vlákien podáme v kapitole *12 Konštrukcia programových vlákien v jazyku C# 3.0*.

V hierarchii vlákien je každé programové vlákno asociované s práve jedným vláknom operačného systému. Vlákno operačného systému je natívnym (nie riadeným) prostriedkom, ktorého manažment riadi jadro operačného systému (a nie virtuálny exekučný systém). V prípade n-vláknových riadených aplikácií bude vo vrstve operačného systému existovať n natívnych vlákien. S natívnym vláknom je spojená špeciálna entita, ktorá zaznamenáva štatistické údaje o životnom cykle vlákna.

Na strojovej úrovni sú vlákna operačného systému mapované na hardvérové vlákna. Hardvérové vlákno je strojový mechanizmus, ktorý umožňuje simultánnu exekúciu mikroinštrukcií strojového kódu. Pritom platí, že každé exekučné jadro procesora smie v rovnakom čase spracúvať jedno hardvérové vlákno. Keďže medzi hardvérovými vláknami a exekučnými jadrami procesora existuje relácia typu 1:1,

pri n-jadrovom procesore bude môcť byť paralelne realizovaných n hardvérových vlákien.

Typológiu vlákien znázorňuje obr. 5.1.



Obr. 5.1: Typológia vlákien

Programové vlákna sa delia na nasledujúce 2 skupiny:

1. Programové vlákna v popredí[11].
2. Programové vlákna v pozadí[12].

Rozdiel medzi vláknami v popredí a vláknami v pozadí spočíva vo vplyve, aký majú na životný cyklus riadenej aplikácie. Riadená aplikácia beží dovtedy, dokiaľ existuje aspoň jedno programové vlákno v popredí, ktoré je zaneprázdnené vykonávaním programových príkazov. I jedno vlákno v popredí dokáže „udržať“ riadenú aplikáciu pri živote, čo je vlastnosť, ktorú vlákna v pozadí nemajú. Bez ohľadu na to, koľko existuje vlákien v pozadí, ak neexistuje žiadne aktívne vlákno v popredí, exekúcia

[11] Programové vlákna v popredí sa v origináli nazývajú „foreground threads“.
[12] Programové vlákna v pozadí sa v origináli nazývajú „background threads“.

riadenej aplikácie sa končí. Ak riadená aplikácia obsahuje 1 vlákno v popredí a n vlákien v pozadí, tak činnosť všetkých vlákien v pozadí sa skončí vtedy, keď sa terminuje exekúcia príkazov na vlákne v popredí.

Implicitne sú všetky riadené aplikácie jednovláknové, pričom primárne programové vlákno je vždy vláknom v popredí.

6 Technický rozbor procesorových architektúr počítačových systémov

V tejto kapitole podáme technicky orientovaný výklad pracovných modelov rôznych typov mikroprocesorových architektúr s rozličným stupňom podpory paralelného spracovania výpočtových operácií. Pritom sa budeme koncentrovať predovšetkým na nasledujúce typy počítačových systémov:

1. Jednojadrové procesory architektúry IA-32[13] bez technológie HT[14].
2. Jednojadrové procesory architektúry IA-32 s technológiou HT.
3. Viacjadrové procesory architektúry IA-32/Intel 64 bez technológie HT.
4. Viacjadrové procesory architektúry IA-32/Intel 64 s technológiou HT.
5. Viacprocesorové stroje obsahujúce množinu jednojadrových procesorov architektúry IA-32 bez technológie HT.
6. Viacprocesorové stroje obsahujúce množinu jednojadrových procesorov architektúry IA-32 s technológiou HT.
7. Viacprocesorové stroje obsahujúce množinu viacjadrových procesorov architektúry IA-32/Intel 64 bez technológie HT.
8. Viacprocesorové stroje obsahujúce množinu viacjadrových procesorov architektúry IA-32/Intel 64 s technológiou HT.

6.1 Jednojadrové procesory architektúry IA-32 bez technológie HT

Do kategórie jednojadrových procesorov 32-bitovej architektúry IA-32 patria superskalárne procesory s jedným exekučným jadrom, ktoré implementujú

[13] IA-32 je skratka pre 32-bitové mikroprocesorové architektúry spoločnosti Intel (Intel Architecture 32).

[14] HT je skratka pre technológiu Hyper-Threading, ktorá umožňuje efektívnejšiu exekúciu viacvláknových aplikácií.

paralelné spracovanie mikroinštrukcií v jednom hodinovom cykle[15]. Prvým procesorom tejto triedy bol Intel Pentium uvedený v roku 1993.

Jednojadrový procesor obsahuje tieto funkcionálne bloky:

1. **Vyrovnávacie (cache) pamäte rôznych úrovní.** Hlavnou úlohou vyrovnávacích pamätí je minimalizovať latenciu, ktorá vzniká pri transporte dát a programových inštrukcií z operačnej pamäte do procesora. Jednotlivé úrovne vyrovnávacích pamätí sa vo všeobecnosti označujú systémom L_1, L_2, ..., L_n. Jednojadrový procesor obsahuje spravidla dve úrovne rýchlych vyrovnávacích pamätí (L_1 a L_2). So vzrastajúcim ordinálnym číslovaním úrovní vyrovnávacích pamätí sa spájajú dva efekty: zvyšovanie alokačnej kapacity a zvyšovanie prístupovej doby. Vyrovnávacia pamäť L_1 je pri niektorých typoch jednojadrových procesorov výhradne dátového charakteru (môže obsahovať len dáta, nie programové inštrukcie). Iné typy jednojadrových procesorov obsahujú segmentovanú vyrovnávaciu pamäť L_1. Segmentácia rozdeľuje pamäť L_1 na dve časti, z ktorých jedna je určená na uchovanie dát a druhá slúži na archiváciu programových inštrukcií. Vyrovnávacia pamäť druhej úrovne (L_2) je unifikovaná, takže v nej môžu byť uložené dáta spoločne s programovými inštrukciami (spomenutá unifikácia platí aj pre vyrovnávaciu pamäť tretej úrovne – L_3 – exkluzívne začlenenú do serverových jednojadrových procesorov).

   V okamihu, keď procesor potrebuje uskutočniť istú operáciu s dátami, zistí, či sa tieto dáta nachádzajú v jednej z vyrovnávacích pamätí. Ak áno, dáta sú okamžite dosiahnuteľné, a preto môžu byť ihneď použité pri realizácii výpočtových operácií. Dôležité je, že vyrovnávacie pamäte sú prehľadávané po úrovniach s tým, že ako prvá je vždy analyzovaná pamäť L_1. Princíp detekcie dát je logický, pretože procesor najskôr skenuje vyrovnávaciu pamäť s najmenšou lokalitou. Ak sú dáta nájdené v pamäti L_1, sú

[15] Technika paralelného spracovania mikroinštrukcií v jednom hodinovom cykle predstavuje nízkoúrovňový paralelizmus (ILP, Instruction Level Parallelism). Každý procesor, ktorý je schopný implementovať takýto nízkoúrovňový paralelizmus, sa označuje termínom superskalárny.

bezprostredne použité[16]. Ak sa dáta vo vyrovnávacej pamäti 1. úrovne nenachádzajú[17], procesor analyzuje pamäť L_2. V prípade, že dáta sú identifikované v tejto úrovni vyrovnávacej pamäte, prenesú sa do pamäte L_1 a vzápätí sú priamo použité procesorom. Pri 2-úrovňovej skladbe vyrovnávacích pamätí neexistuje žiadna ďalšia vyrovnávacia pamäť, takže ak požadované dáta nebudú situované ani v pamäti L_2, je nutné vykonať transport do operačnej pamäte a získať dáta z nej. Vzhľadom na to, že latencia je pri práci s operačnou pamäťou vskutku signifikantná, z pohľadu procesora je operácia „získať dáta z operačnej pamäte" ponímaná ako veľmi pomalá (so spotrebou niekoľko tisíc až niekoľko desaťtisíc hodinových cyklov procesora).

2. **Exekučné jadro procesora.** Exekučné jadro procesora tvoria viaceré exekučné prostriedky, ku ktorým patrí:

   - Aritmeticko-logická jednotka na spracovanie operácií s celými číslami.
   - Aritmeticko-logická jednotka na spracovanie operácií s reálnymi číslami.
   - Jednotka realizujúca hardvérové vetvenie tokov mikroinštrukcií.

3. **Lokálny správca prerušení: L-APIC (Local Advanced Programmable Interrupt Controller).** Správca L-APIC prijíma prerušenia od interných prostriedkov jednojadrového procesora a od externého správcu programových prerušení (I/O-APIC). Všetky prijaté prerušenia sú transponované exekučnému jadru procesora, pričom riadia jeho činnosť.

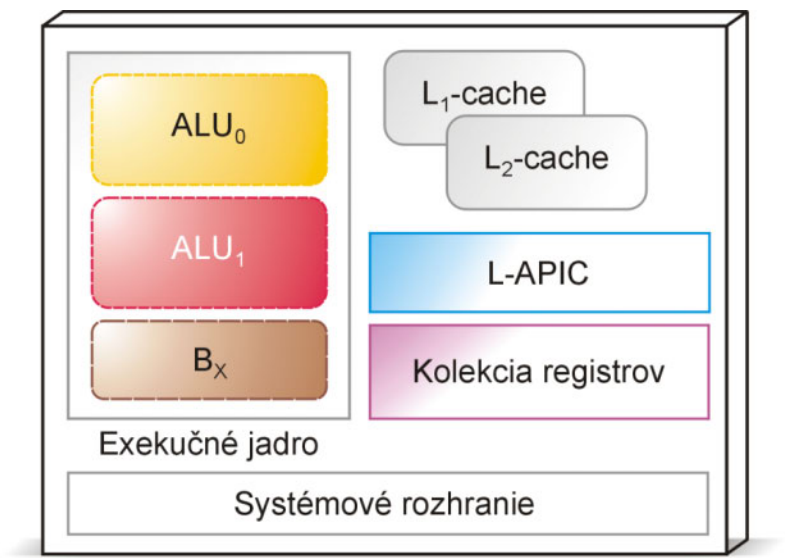
4. **Registre.** Jednojadrové procesory architektúry IA-32 v sebe integrujú kolekciu registrov. Do registrov sú ukladané kvantá dát s vysokou frekvenciou používania. Prístup do registrov je veľmi rýchly, takže ak sú požadované kvantá dát nájdené už v registroch, úplne sa eliminuje nutnosť

[16] Tomuto javu sa vraví „dátový zásah", angl. „cache hit".
[17] Tento jav je známy ako „dátový omyl", angl. „cache miss".

skenovať ďalšie typy počítačových pamätí. Kolekcia registrov je zložená z množstva registrov rozličných typov. Napríklad dátové registre sú určené na archiváciu celočíselných alebo reálnych kvánt dát. Adresové registre zase slúžia na uchovanie pamäťových adries. Procesory Intel IA-32 obsahujú tieto typy procesorových registrov: všeobecné registre (na uchovanie dát a adries), segmentové registre (sú nositeľmi 16-bitových segmentových selektorov[18]), EFLAGS registre (registre určené na zisťovanie stavu vykonávaného programu a obmedzenú kontrolu procesora) a EIP register (tento register obsahuje 32-bitový smerník identifikujúci mikroinštrukciu, ktorá má byť vykonaná v ďalšom kroku).

5. **Systémové rozhranie.** Systémové rozhranie spája procesor so systémovou zbernicou, a teda so všetkými ostatnými hardvérovými komponentmi počítačového systému.

Schematickú konštrukciu jednojadrového procesora architektúry IA-32 ukazuje obr. 6.1.



Obr. 6.1: Jednojadrový procesor architektúry IA-32

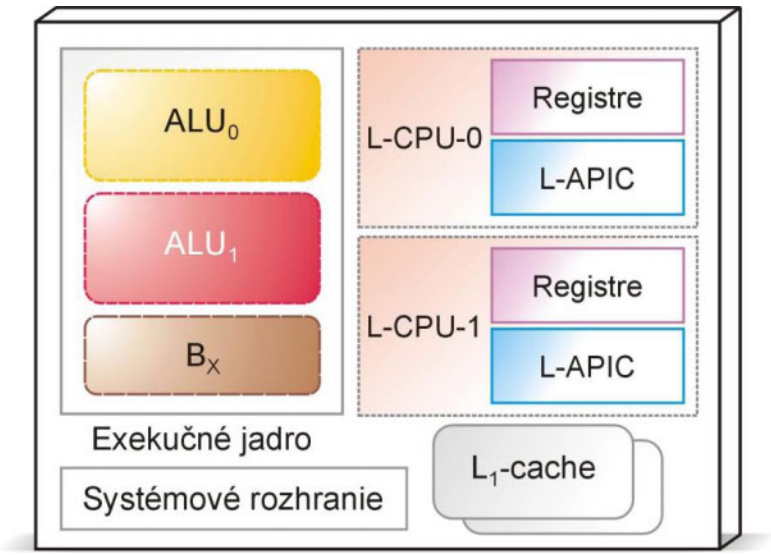
[18] Segmentový selektor je smerník ukazujúci na pamäťový segment.

Na obr. 6.1 vidíme, že procesor integruje:

- aritmeticko-logickú jednotku na spracovanie operácií s celými číslami (ALU_0),
- aritmeticko-logickú jednotku na spracovanie operácií s reálnymi číslami (ALU_1),
- jednotku realizujúcu hardvérové vetvenie tokov mikroinštrukcií (B_X),
- 2-úrovňovú vyrovnávaciu pamäť (L_1-cache, L_2-cache),
- lokálneho správcu prerušení,
- kolekciu registrov
- a systémové rozhranie.

6.2 Jednojadrové procesory architektúry IA-32 s technológiou HT

Technológia Intel Hyper-Threading (HT) implementuje do jednojadrových procesorov architektúry Intel IA-32 mechanizmus simultánneho spracovania viacerých programových vlákien. Simultánne spracovanie programových vlákien je možné vďaka duplikácii prostriedkov procesora, ktoré reprezentujú jeho architektonický stav. K prostriedkom definujúcim architektonický stav procesora patria predovšetkým registre a L-APIC. Ak dôjde k duplikácii architektonického stavu procesora, z pohľadu operačného systému a aplikačného softvéru sa fyzicky jednojadrový procesor začne javiť ako množina logických procesorov. Pokiaľ je jednojadrový procesor vybavený technológiou HT, je ekvivalentný dvom logickým procesorom. Primárnou podstatou technológie HT je zabezpečenie maximálneho využitia výpočtovej kapacity jednojadrového procesora simultánnym spracovaním mikroinštrukcií, ktoré patria rozdielnym programovým vláknam vykonávaných na samostatných logických procesoroch. Pri analýze procesorov s HT technológiou je dôležité podotknúť, že oba logické procesory zdieľajú jedno exekučné jadro fyzického procesora. Procesor s HT technológiou teda nie je viacjadrovým procesorom v pravom slova zmysle, pretože obsahuje iba jedno exekučné jadro.



Obr. 6.2: Jednojadrový procesor architektúry IA-32 s HT technológiou

Na obr. 6.2 je zaznamenaná organizácia jednojadrového procesora s HT technológiou. Je zrejmé, že fyzický procesor obsahuje 2 logické procesory s identifikátormi L-CPU-0 a L-CPU-1. Oba logické procesory zdieľajú exekučné jadro fyzického procesora, vyrovnávacie pamäte a systémové rozhranie.

Keďže existujú 2 logické procesory, ktoré súperia o 1 exekučné jadro, je nutné zaviesť mechanizmus, ktorý bude garantovať efektívnu implementáciu simultánneho viacvláknového spracovania. Tento mechanizmus je založený na injektovaní mikroinštrukcií z rôznych programových vlákien, ktoré sú súbežne spracúvané na logických procesoroch. Predpokladajme nasledujúci pracovný model:

1. Pri spustení 2-vláknovej aplikácie sú jednotlivé vlákna rovnomerne rozložené na 2 logické procesory.

2. Exekúcia je zahájená 1. vláknom, ktoré je spracúvané na 1. logickom procesore. Vzhľadom na to, že 1. logický procesor v tejto chvíli úplne okupuje exekučné jadro fyzického procesora, 2. vlákno delegované na 2. logický procesor nezaznamenáva žiaden exekučný progres.

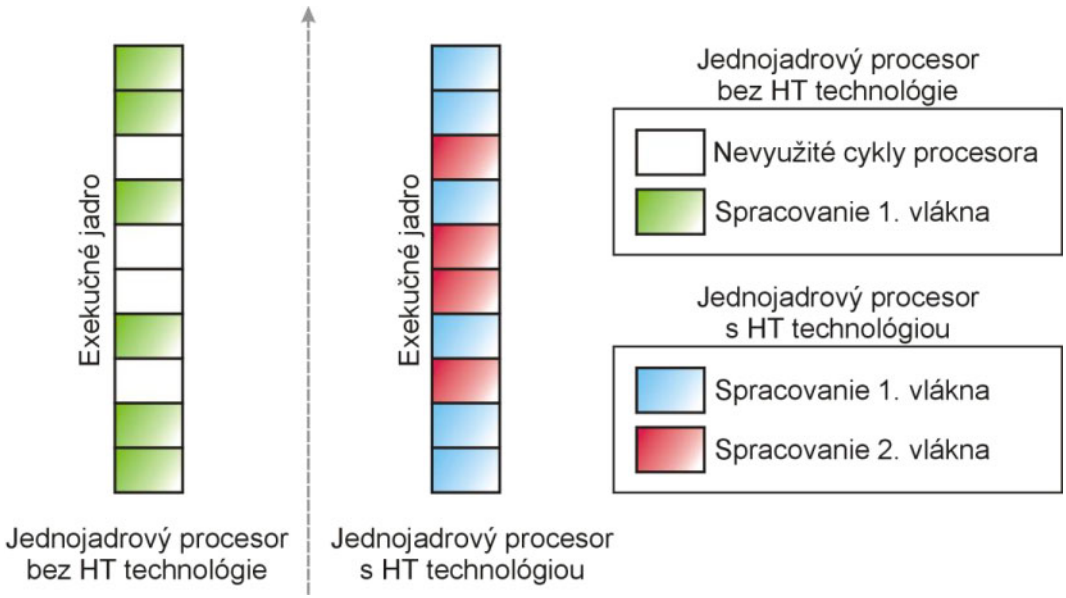
3. V životnom cykle 1. vlákna sa objaví požiadavka na časovo intenzívnu operáciu, napríklad požiadavka na načítanie súpravy dát. V závislosti od lokality dát môže spracovanie tejto požiadavky trvať desiatky, stovky, alebo aj tisícky hodinových cyklov exekučného jadra fyzického procesora. Aby počas tohto časového intervalu nedošlo k neželanému plytvaniu výpočtovou kapacitou exekučného jadra fyzického procesora, technológia HT pozastaví 1. vlákno spracúvané na 1. logickom procesore a zaháji exekúciu 2. vlákna na 2. logickom procesore. Tak je exekučné jadro fyzického procesora zaťažené realizovaním mikroinštrukcií 2. programového vlákna.

4. Keďže časovo intenzívne operácie z pohľadu práce exekučného jadra fyzického procesora[19] sa v životných cykloch programových vlákien vykonávaných na logických procesoroch objavujú s pomerne veľkou frekvenciou, technológia HT dokáže efektívne prepínať medzi programovými vláknami, čoho dôsledkom je aktívne udržiavanie exekučného jadra procesora v maximálne vyťaženom stave.

Na obr. 6.3 sa nachádza vizuálna komparácia pracovných modelov dvoch jednojadrových procesorov: jeden procesor HT technológiu nepodporuje, kým druhý áno. Analýza sa koncentruje na určité štádium životného cyklu aplikácie, počas ktorého sledujeme pracovnú efektívnosť exekučného jadra procesora. Pre zjednodušenie uvažujeme o veľmi malom štádiu životnosti aplikácie, ktoré na hardvérovej úrovni alokuje maximálne 10 cyklov exekučného jadra procesora. Ak sa počas tohto štádia vyskytnú časovo intenzívne operácie s významnou latenciou, procesor bez implementovanej HT technológie generuje množinu nevyužitých cyklov. Nevyužité cykly sú cykly, v priebehu ktorých nie je procesor zamestnaný žiadnou činnosťou (často vravíme, že procesor sa nachádza v nečinnom stave). Pri procesore bez HT technológie tvoria nevyužité cykly v danom konkrétnom prípade 40 % všetkých analyzovaných cyklov.

[19] K časovo intenzívnym operáciám patrí okrem načítania či uloženia dát aj čakanie na výsledok predchádzajúcej operácie, čakanie na opätovné načítanie dát do vyrovnávacích pamätí pri dátovom omyle, alebo tiež penalizácie, ktoré vznikajú dôsledkom chybných predikcií budúcich exekučných vetiev mikroinštrukcií.

Na druhej strane, procesor so začlenenou podporou pre HT technológiu dokáže svoje výpočtové kapacity naplno obsadiť. Ak je potrebné čakať na spracovanie inštrukcií jedného vlákna, exekučnému jadru sú ponúknuté inštrukcie druhého vlákna. Tým dochádza k eliminácii časových intervalov, počas ktorých procesor generuje nevyužité pracovné cykly.



Obr. 6.3: Komparácia pracovných modelov jednojadrových procesorov bez a s HT technológiou

Prvým procesorom s podporou technológie HT sa stal Intel Xeon, určený pre nasadenie v serverových pracovných staniciach. Prvým procesorom s podporou technológie HT pre segment finálnych domácich a firemných používateľov bol Intel Pentium 4. Spoločnosť Intel uvádza, že zvýšenie výkonnosti softvérových aplikácií na jednojadrových procesoroch s HT technológiou sa v priemere pohybuje v rozsahu 10 až 30 percent, podľa toho, do akej miery sú aplikácie optimalizované pre simultánne spracovanie viacerých tokov programových inštrukcií.

6.3 Viacjadrové procesory architektúry IA-32/Intel 64 bez technológie HT

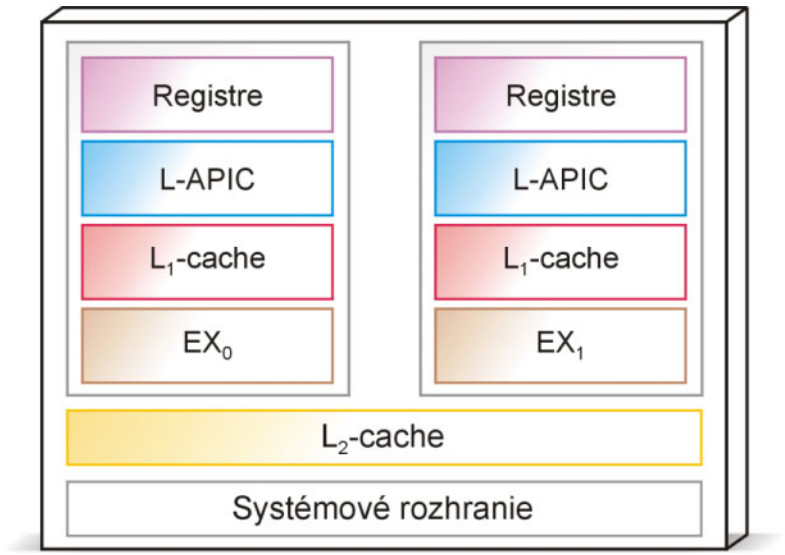
Viacjadrový procesor je procesor, ktorý v jednom fyzickom balení obsahuje n exekučných jadier, pričom $n \geq 2$. Každé z osadených exekučných jadier je schopné spracovať jedno hardvérové vlákno mikroinštrukcií. Exekučné jadro pozostáva z funkcionálnych blokov, ktoré zabezpečujú vykonanie mikroinštrukcií hardvérového vlákna. K týmto blokom patria výpočtové aritmeticko-logické jednotky, jednotky na vetvenie tokov mikroinštrukcií, procesorové registre, L-APIC a vyrovnávacia pamäť 1. úrovne (L_1-cache). Vyrovnávacia pamäť 2. úrovne (L_2-cache) je buď zdieľaná všetkými exekučnými jadrami procesora, alebo dedikovaná, čiže výhradne pridelená každému exekučnému jadru procesora. Spravidla však platí pravidlo, že ak je nejaká vyrovnávacia pamäť zdieľaná, tak ide o vyrovnávaciu pamäť poslednej úrovne, tzv. LLC (Last Level Cache). Ak je pamäťou LLC vyrovnávacia pamäť 2. úrovne (L_2-cache), môže byť zdieľaná ona. Naopak, ak procesor obsahuje vyrovnávaciu pamäť 3. úrovne (L_3-cache), bude zdieľaná táto pamäť.

Počet exekučných jadier viacjadrových procesorov môžeme ohraničiť exponenciálnou funkciou $f(n) = 2^n$, pre $n \in < 1, \infty)$. Prirodzene, maximálny počet exekučných jadier je obmedzený súčasnými hardvérovými technológiami. Nárast počtu exekučných jadier môže byť symetrický (presne modelovaný funkciou $f(n) = 2^n$) alebo asymetrický (iba ohraničený funkciou $f(n) = 2^n$). Dôsledkom tejto skutočnosti je existencia 2-jadrových, 3-jadrových a 4-jadrových procesorov na trhu hardvérových komponentov. Najrozšírenejšie viacjadrové procesory spoločnosti Intel bez technológie HT sú tieto:

- Intel Core 2 Duo, 2-jadrový procesor.
- Intel Core 2 Quad, 4-jadrový procesor.

Pre úplnosť dodajme, že 3-jadrové procesory Phenom X3 produkuje spoločnosť Advanced Micro Devices (AMD).

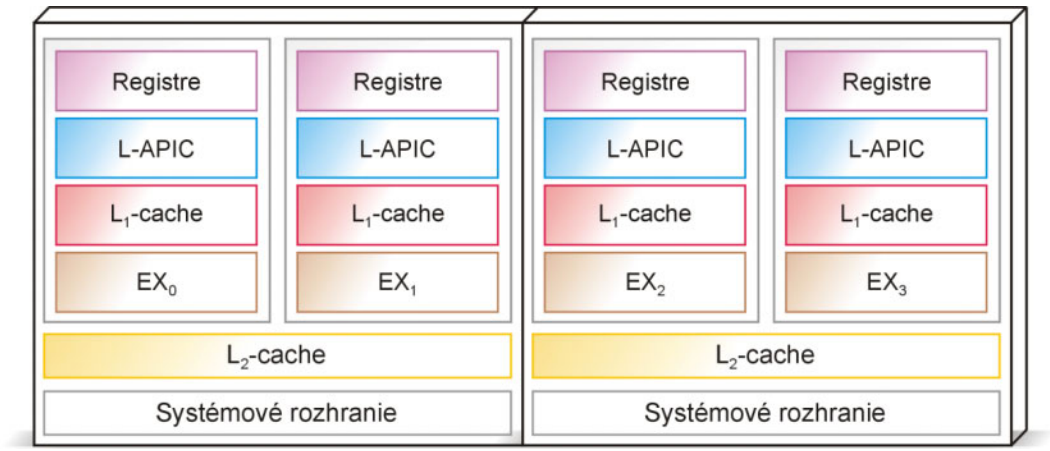
Konštrukciu 2-jadrového procesora Intel Core 2 Duo môžeme vidieť na obr. 6.4.



Obr. 6.4: Intel Core 2 Duo, 2-jadrový procesor bez technológie HT

Viacjadrový procesor Intel Core 2 Duo obsahuje 2 exekučné jadrá, pričom každé z nich združuje tieto súčasti: registre, L-APIC, L_1-cache a exekučný stroj (EX_0, resp. EX_1). Vyrovnávacia pamäť L_2-cache je zdieľaná obidvomi exekučnými jadrami. Podobne je zdieľané aj systémové rozhranie.

Konštrukciu 4-jadrového procesora Intel Core 2 Quad ukazuje obr. 6.5.

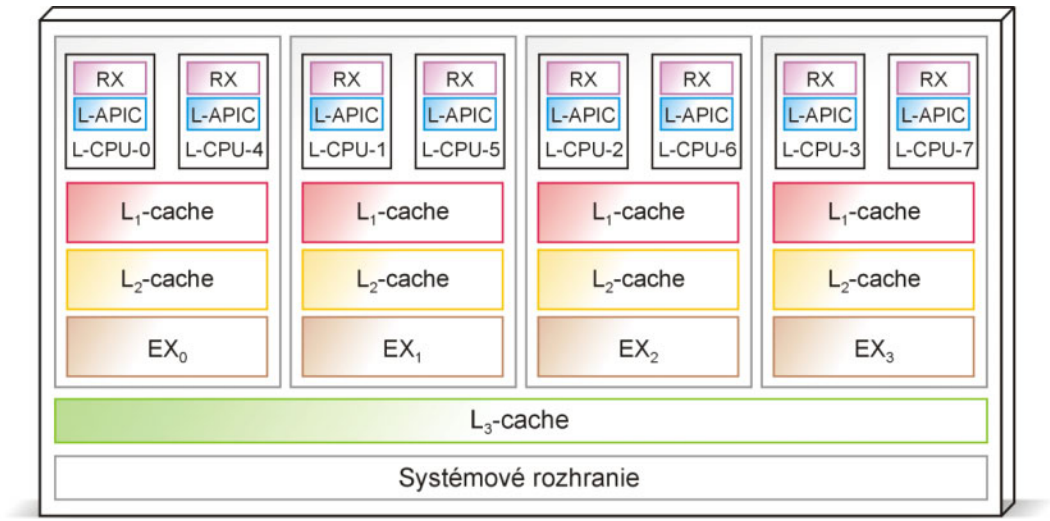


Obr. 6.5: Intel Core 2 Quad, 4-jadrový procesor bez technológie HT

Viacjadrový procesor Intel Core 2 Quad zahŕňa 4 exekučné jadrá s rovnakým počtom exekučných strojov (EX_0, EX_1, EX_2 a EX_3). Napriek tejto skutočnosti nejde o natívny 4-jadrový procesor, ale o prepojenie dvoch 2-jadrových procesorov. Každá dvojica exekučných jadier zdieľa vyrovnávaciu pamäť 2. úrovne, podobne ako aj systémové rozhranie.

6.3 Viacjadrové procesory architektúry IA-32/Intel 64 s technológiou HT

Ak viacjadrový procesor implementuje technológiu HT, tak každé exekučné jadro predstavuje dvojicu logických procesorov. Prvým viacjadrovým procesorom s implementovanou technológiou HT bol Intel Pentium Extreme Edition, uvedený v roku 2005. Tento procesor obsahoval 2 exekučné jadrá s HT technológiou, takže softvérové aplikácie mohli v súčinnosti s operačným systémom využívať 4 logické procesory. Najmodernejším viacjadrovým procesorom s technológiou HT je Intel Core i7. Tento procesor je natívny 4-jadrový procesor (integruje 4 exekučné jadrá v jednom fyzickom balení) a vďaka začlenenej technológii HT naň smieme nahliadať ako na súpravu 8 logických procesorov. Grafický model viacjadrového procesora Intel Core i7 je znázornený na obr. 6.6.



Obr. 6.6: Intel Core i7, 4-jadrový procesor s technológiou HT

Z obr. 6.6 je zrejmé, že mikroprocesorová architektúra Intel Core i7 sa značne odlišuje od predchádzajúcej architektúry Intel Core 2 Duo a Intel Core 2 Quad. Procesor Intel Core i7 obsahuje 4 exekučné jadrá s rovnakým počtom exekučných strojov (EX_0, EX_1, EX_2 a EX_3). Keďže funkcionálne bloky definujúce architektonický stav exekučného jadra boli v každom jadre zduplikované, jedno exekučné jadro obsahuje 2 logické procesory, ktoré zdieľajú výpočtové kapacity exekučného stroja. Každé exekučné jadro má dedikované 2 úrovne vyrovnávacej pamäte (L_1-cache a L_2-cache). Všetky exekučné jadrá využívajú zdieľanú L_3-cache, podobne ako systémové rozhranie.

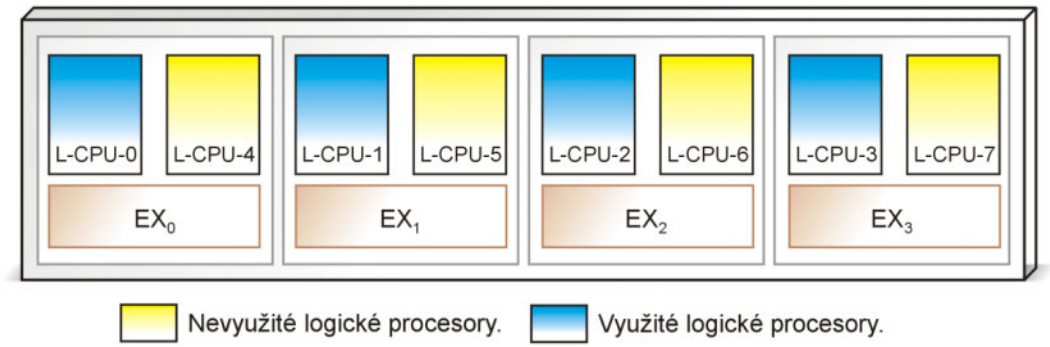
Procesor Intel Core i7 sa skladá z 8 logických procesorov. Všimnime si vzor, ktorý je použitý pri identifikovaní jednotlivých logických procesorov (tab. 6.1).

Exekučné jadro	Identifikátory logických procesorov
1. exekučné jadro	L-CPU-0
	L-CPU-4
2. exekučné jadro	L-CPU-1
	L-CPU-5
3. exekučné jadro	L-CPU-2
	L-CPU-6
4. exekučné jadro	L-CPU-3
	L-CPU-7

Tab. 6.1: Vzor identifikácie logických procesorov
na exekučných jadrách procesora Intel Core i7

Tento identifikačný vzor je pozoruhodný, pretože reflektuje snahy operačného systému o efektívnejšie plánovanie úloh pre logické procesory v prípade, ak je počet programových vlákien softvérovej aplikácie menší, alebo nanajvýš rovný počtu exekučných jadier viacjadrového procesora s podporou technológie HT.

Základný princíp spočíva v tom, aby boli rôzne úlohy delegované na nečinné logické procesory, ktoré ležia na rôznych exekučných jadrách viacjadrového procesora. Predpokladajme, že na procesore Intel Core i7 s 8 logickými procesormi je spracúvaná 4-vláknová aplikácia. Keďže dvojice logických procesorov musia súperiť o jedno exekučné jadro, najefektívnejšie je delegovať 4 programové vlákna softvérovej aplikácie na 4 logické procesory, ktoré sa nachádzajú na rôznych exekučných jadrách procesora. Túto situáciu zachytáva obr. 6.7.



Obr. 6.7: Optimálne využitie logických procesorov

Vzťahy medzi programovými vláknami aplikácie a logickými procesormi môžeme charakterizovať takto:

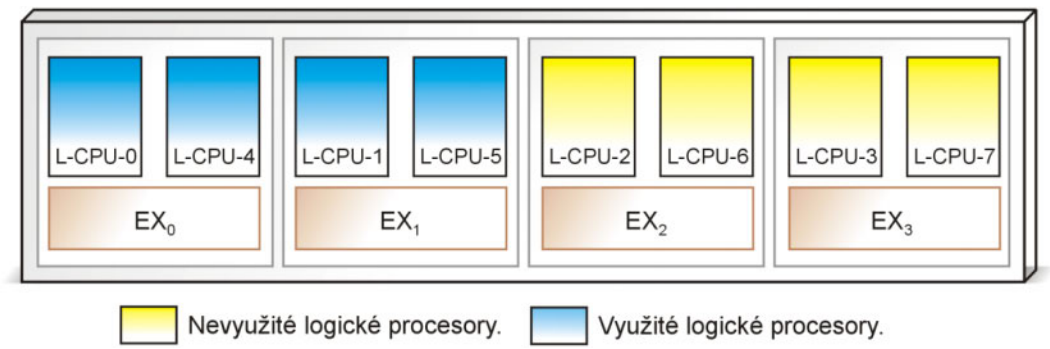
- 1. programové vlákno je spracúvané na logickom procesore L-CPU-0.
- 2. programové vlákno je spracúvané na logickom procesore L-CPU-1.
- 3. programové vlákno je spracúvané na logickom procesore L-CPU-2.
- 4. programové vlákno je spracúvané na logickom procesore L-CPU-3.

Vzhľadom na to, že v každom exekučnom jadre je aktívny práve jeden logický procesor, sú výpočtové kapacity príslušného exekučného jadra pridelené výhradne len tomuto logickému procesoru. Nedochádza tak k žiadnemu zdieľaniu exekučného stroja jadra, čo je za daných okolností optimálne riešenie.

Výkonnosť 4-vláknovej softvérovej aplikácie by však istotne poklesla, keby plánovač úloh operačného systému vytvoril takúto koreláciu medzi programovými vláknami a logickými procesormi (obr. 6.8):

- 1. programové vlákno je spracúvané na logickom procesore L-CPU-0.
- 2. programové vlákno je spracúvané na logickom procesore L-CPU-4.
- 3. programové vlákno je spracúvané na logickom procesore L-CPU-1.
- 4. programové vlákno je spracúvané na logickom procesore L-CPU-5.

1. a 2. programové vlákno zdieľajú exekučný stroj EX_0 1. exekučného jadra procesora, zatiaľ čo 3. a 4. programové vlákno okupujú exekučný stroj EX_1 2. exekučného jadra procesora. Pretože logické procesory L-CPU-0 a L-CPU-4 musia zdieľať exekučný stroj EX_0, efektívnosť ich práce nebude taká vysoká, ako keby každý z logických procesorov využíval dedikovaný exekučný stroj. Samozrejme, to isté platí aj pre činnosť logických procesorov L-CPU-1 a L-CPU-5.

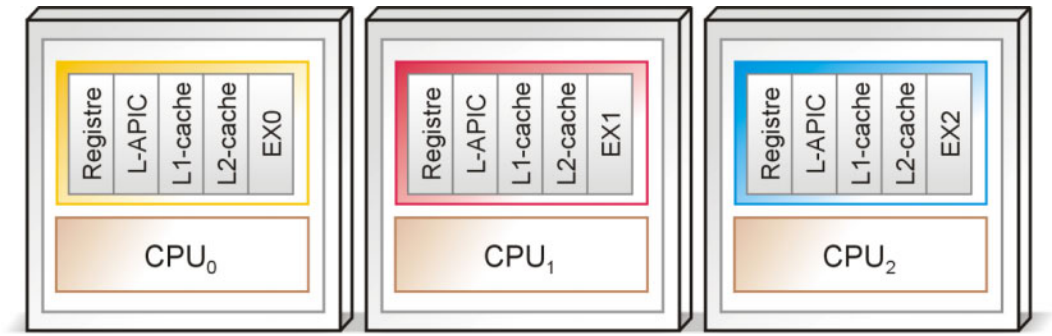


Obr. 6.8: Suboptimálne využitie logických procesorov

6.4 Viacprocesorové architektúry IA-32/Intel 64

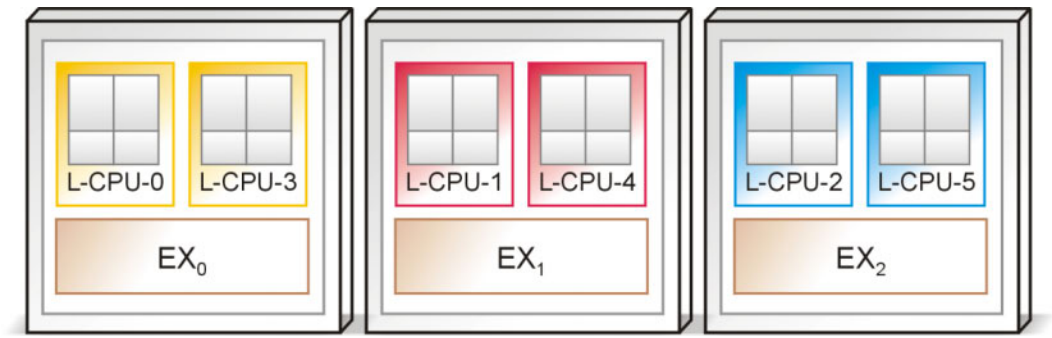
Ak počítač uchováva viacero fyzických procesorov, tak ide o paralelný systém s viacprocesorovou architektúrou. V rámci našej technickej analýzy sa sústredíme na tieto typy viacprocesorových systémov:

1. **Viacprocesorové stroje obsahujúce množinu jednojadrových procesorov architektúry IA-32 bez technológie HT.** Tento typ viacprocesorových systémov bol v minulosti najrozšírenejší. Jeho produkcia nebola príliš komplikovaná, pretože jednotlivé jednojadrové procesory pôsobili ako samostatné výpočtové jednotky so vzájomným prepojením. Na obr. 6.9 ukazujeme konštrukciu viacprocesorového systému, ktorý je osadený trojicou samostatných jednojadrových procesorov architektúry IA-32 bez technológie HT.



Obr. 6.9: Viacprocesorový stroj obsahujúci 3 jednojadrové procesory

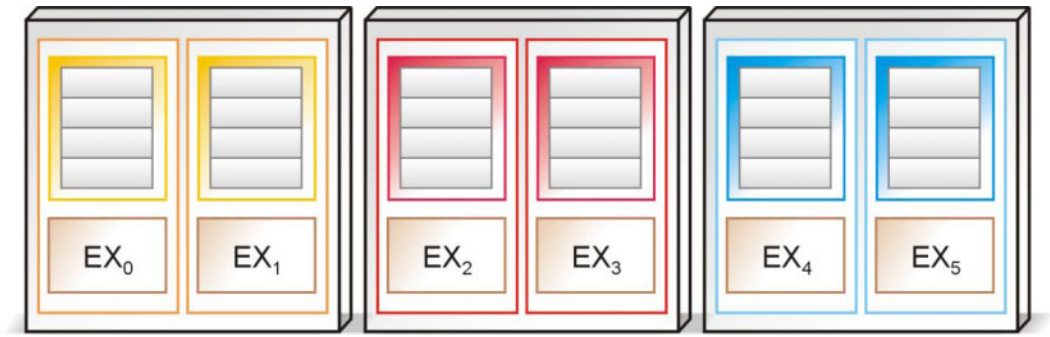
2. **Viacprocesorové stroje obsahujúce množinu jednojadrových procesorov architektúry IA-32 s technológiou HT.** Keď do počítača nainštalujeme n diskrétnych jednojadrových procesorov s technológiou HT, paralelný systém bude obsahovať $n \times 2$ logických procesorov. Na obr. 6.10 je znázornený viacprocesorový systém s 3 fyzickými jednojadrovými procesormi s technológiou HT. Celkovo teda tento viacprocesorový stroj obsahuje 6 logických procesorov.



Obr. 6.10: Viacprocesorový stroj zložený z 3 jednojadrových procesorov s technológiou HT (dovedna 6 logických procesorov)

3. **Viacprocesorové stroje obsahujúce množinu viacjadrových procesorov architektúry IA-32/Intel 64 bez technológie HT.** Za predpokladu, že viacprocesorový systém integruje m viacjadrových procesorov bez technológie HT, pričom každý viacjadrový procesor pozostáva z n

exekučných jadier, môžeme na takomto viacprocesorovom stroji paralelne spracúvať $m \times n$ hardvérových vlákien. Obr. 6.11 predstavuje schematický model viacprocesorového systému, ktorý obsahuje 3 2-jadrové procesory bez technológie HT. Na takomto počítači môže byť súbežne vykonávaná šestica hardvérových vlákien.

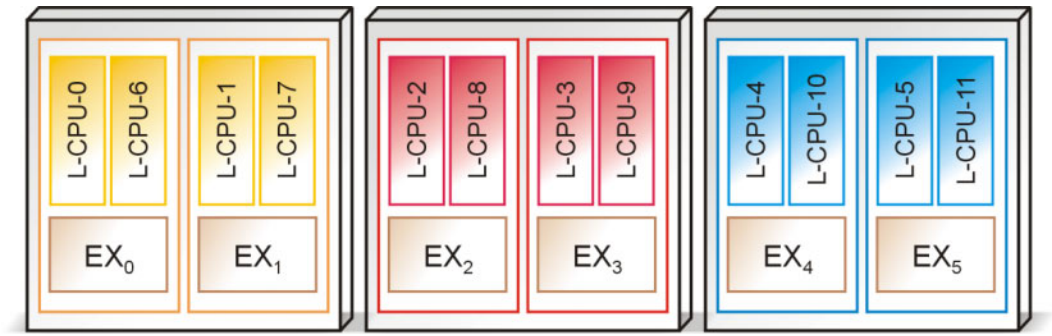


Obr. 6.11: Viacprocesorový stroj zhotovený z 3 2-jadrových procesorov bez technológie HT (dovedna 6 exekučných jadier)

4. **Viacprocesorové stroje obsahujúce množinu viacjadrových procesorov architektúry IA-32/Intel 64 s technológiou HT.** V prípade, ak je viacprocesorový systém vybavený m viacjadrovými procesormi, pričom každý viacjadrový procesor obsahuje n exekučných jadier a podporuje technológiu HT, tak celková výpočtová kapacita je daná $m \times n \times 2$ logickými procesormi. Povedané inak, takýto viacprocesorový stroj je schopný spracovať:

   - $m \times n$ hardvérových vlákien,
   - $m \times n \times 2$ programových vlákien.

   Obr. 6.12 ukazuje viacprocesorový systém, ktorý sme získali poskladaním 3 2-jadrových procesorov s technológiou HT. Tento viacprocesorový systém obsahuje celkom 6 fyzických a 12 logických procesorov.



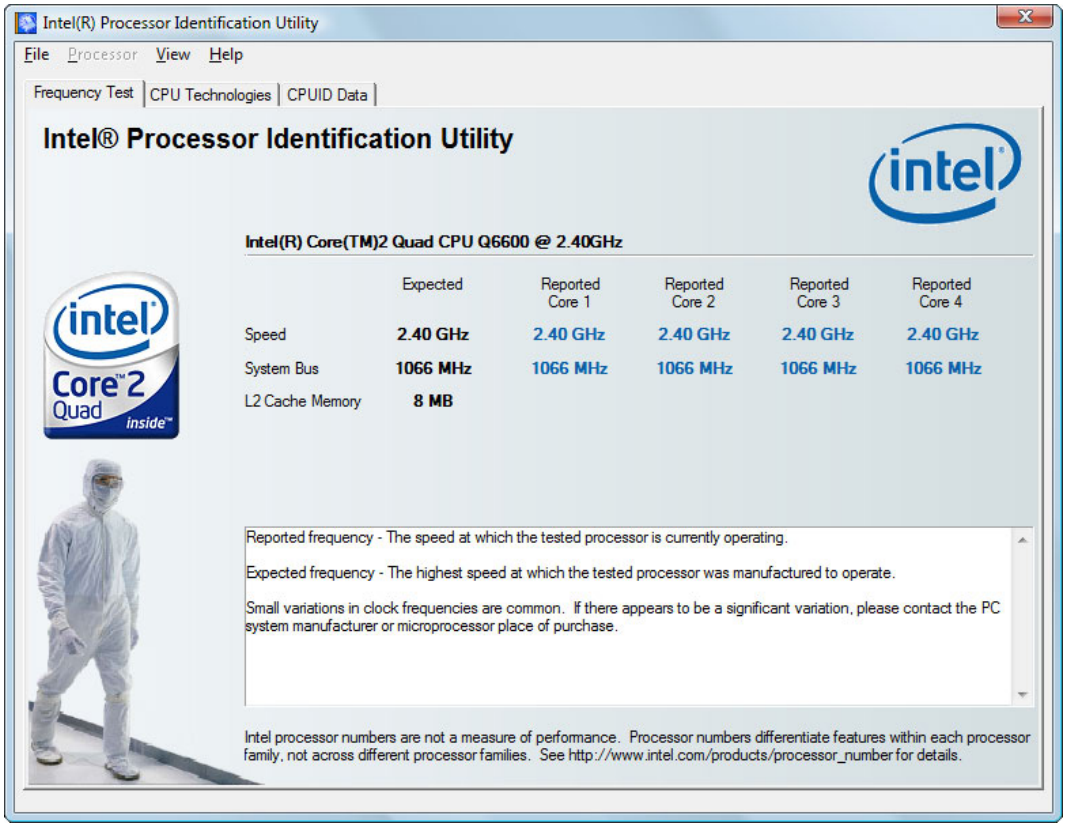
Obr. 6.12: Viacprocesorový stroj skonštruovaný z 3 2-jadrových procesorov s technológiou HT (dovedna 6 fyzických a 12 logických procesorov)

6.5 Softvér na programovú identifikáciu procesorov architektúry Intel IA-32 a Intel 64

Spoločnosť Intel ponúka softvérovým vývojárom a počítačovým používateľom programové vybavenie na zistenie základných aj pokročilých informácií o jedno- a viacjadrových procesoroch architektúr IA-32 a Intel 64, ako aj o technológiách, ktoré tieto procesory podporujú.

Na webovej adrese http://www.intel.com/support/processors/tools/piu/ je k dispozícii na voľné prevzatie softvér **Intel Processor Identification Utility**. V čase písania tejto publikácie bola najaktuálnejšia verzia určená pre operačné systémy Microsoft Windows označená číselným identifikátorom 4.0.

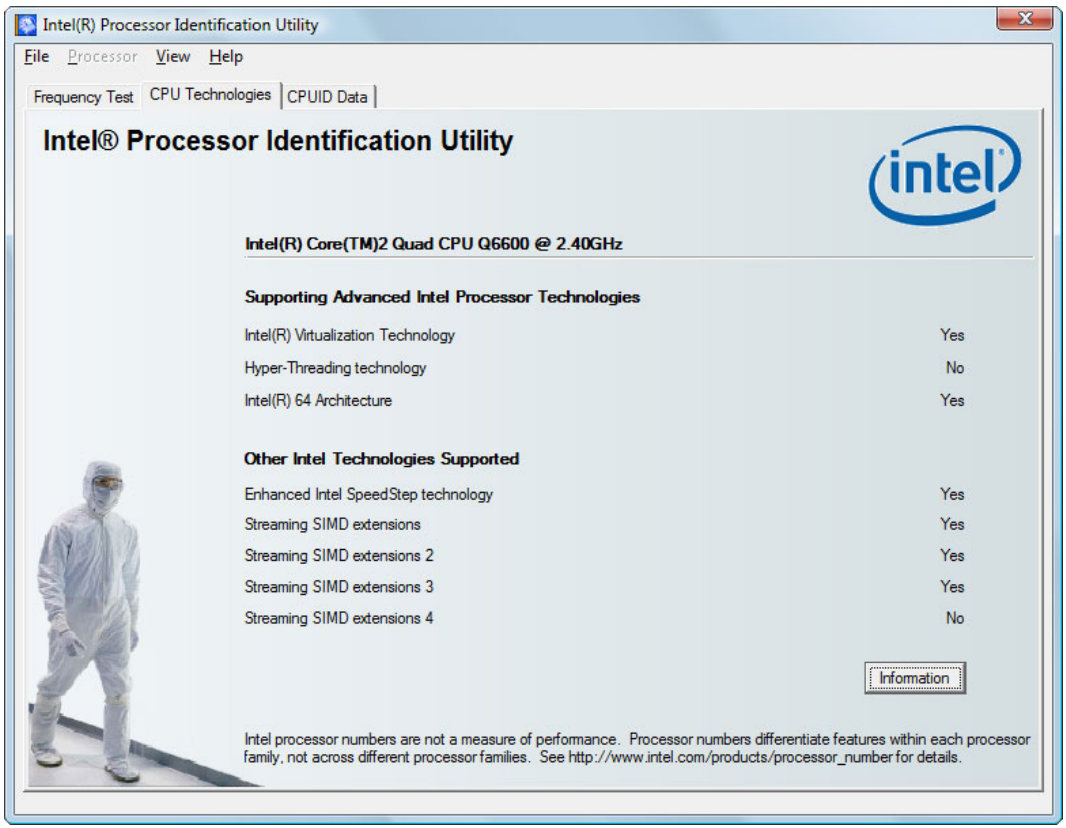
Po spustení programu **Intel Processor Identification Utility** môžeme spoľahlivo analyzovať schopnosti procesora, ktorý je nainštalovaný v našej počítačovej stanici (obr. 6.13).



Obr. 6.13: Softvér **Intel Processor Identification Utility** zisťuje informácie o technických parametroch jedno- a viacjadrových procesorov architektúr IA-32 a Intel 64

Na obr. 6.13 vidíme, že program diagnostikoval prítomnosť 4-jadrového procesora Intel Core 2 Quad Q6600. Maximálna hodinová frekvencia každého exekučného jadra tohto procesora je 2,4 GHz. Frekvencia, s akou prúdia dáta medzi procesorom a systémovou zbernicou, sa rovná 1066 MHz. Vyrovnávacia pamäť druhej úrovne L_2-cache má celkovú alokačnú kapacitu 8 MB (je rozdelená do dvoch 4-megabajtových blokov). Všetky spomenuté údaje nám poskytne program na (implicitne vybratej) záložke **Frequency Test** okamžite po svojom spustení.

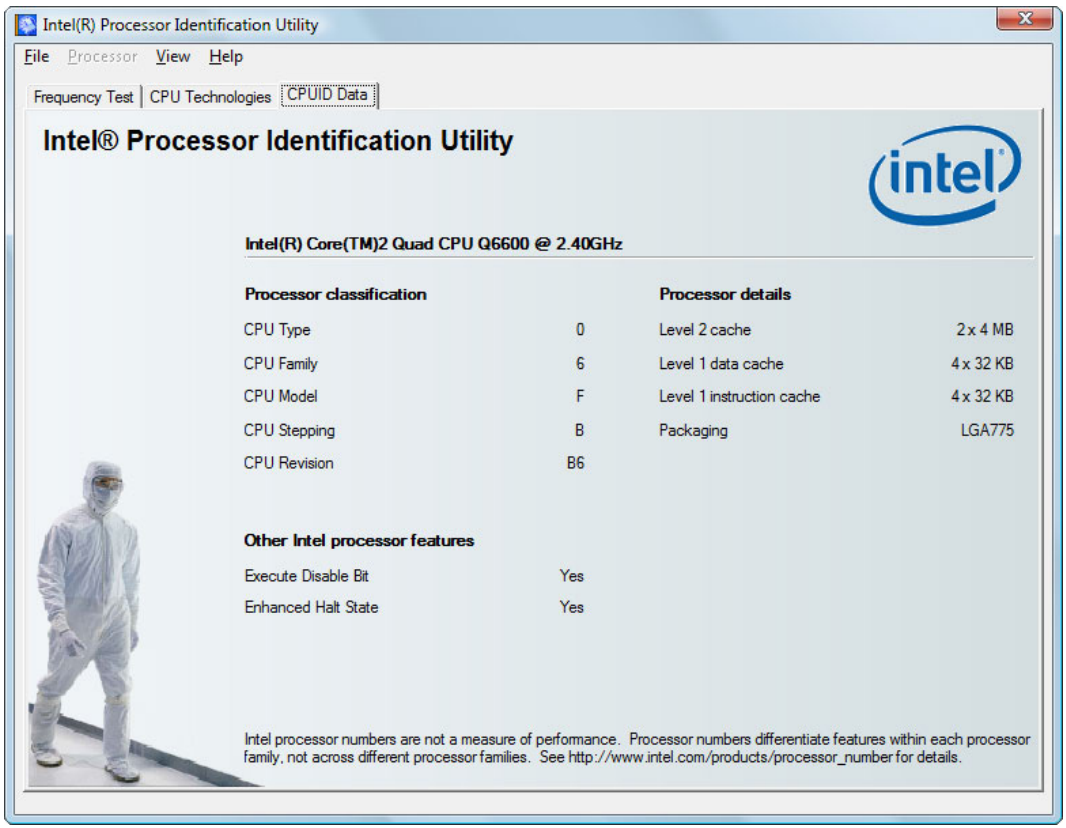
Ak budeme chcieť získať podrobnejšie technické charakteristiky procesora, prepneme sa na ďalšie záložky: **CPU Technologies** (obr. 6.14) a **CPUID Data** (obr. 6.15).



Obr. 6.14: Prehľad podporovaných procesorových technológií

Procesor Intel Core 2 Quad Q6600 podporuje virtualizačnú technológiu Intel Virtualization Technology, podobne ako aj 64-bitovú architektúru Intel 64. Na druhej strane, procesor nepodporuje technológiu Hyper-Threading (HT). Pracovná frekvencia jednotlivých exekučných jadier procesora Intel Core 2 Quad môže byť flexibilne menená prostredníctvom technológie Enhanced Intel SpeedStep Technology, a to podľa aktuálneho pracovného zaťaženia procesora. To napríklad znamená, že pri nízkom zaťažení sa pracovná frekvencia exekučných jadier

procesora smie znížiť z maxima (2,4 GHz) až na 1,6 GHz, čo samozrejme pozitívne vplýva na príkon a ďalšie energetické charakteristiky procesora. Procesor podporuje aj technológie SSE, SSE2 a SSE3[20].



Obr. 6.15: Technické informácie o procesore a vyrovnávacích pamätiach

[20] SSE je skratka z viacslovného spojenia „Streaming SIMD Extensions“. SIMD je zase akronym pre „Single Instruction, Multiple Data“, čo je technológia, ktorá umožňuje paralelne aplikovať jednu aritmetickú operáciu na viaceré kvantá dát. Technológia SSE bola po prvýkrát implementovaná do procesorovej architektúry IA-32 s uvedením procesora Intel Pentium III. Jej prínos spočíva v náraste výkonnosti procesora pri spracúvaní výpočtovo náročných operácií, k akým patrí práca s 2D a 3D počítačovou grafikou, spracovanie obrazových súborov a videosúborov či rozpoznávanie reči.

Z obr. 6.15 je zrejmé, že každé exekučné jadro procesora Intel Core 2 Quad Q6600 obsahuje dedikovanú L_1-cache na úschovu dát i programových inštrukcií, obe s alokačnou kapacitou 32 KB. Pamäť L_2-cache je segmentovaná do dvoch 4-megabajtových blokov, pričom každý blok zdieľajú 2 exekučné jadrá procesora. Okrem zmienených technológií spolupracuje procesor Intel Core 2 Quad Q6600 ešte s dvomi:

- **Execute Disable Bit**: technológia umožňuje chrániť počítač pred vírusmi a škodlivými aplikáciami.

- **Enhanced Halt State**: technológia, ktorá zlepšuje akustické aspekty procesora znížením požiadaviek na príkon.

7 Paralelné programovanie a paralelné objektovo orientované programovanie (POOP)

Podstatou vývoja paralelného počítačového softvéru je analýza, návrh a implementácia paralelných aplikácií, čiže aplikácií, ktoré uskutočňujú súbežné spracovanie programových činností. Aj keď paralelné programovanie ako jedna z paradigiem programovania existuje vo sfére počítačových vied viac desaťročí, práve signifikantný progres hardvérových technológií spôsobil, že sa princípy paralelného programovania dostávajú opäť do popredia.

Počítačové vedy diferencujú viacero paradigiem programovania, ktoré sú založené na paralelnom programovaní. Produktom paralelného programovania sú aplikácie pre finálnych používateľov, ktoré pracujú na diskrétnych počítačových staniciach. Distribuované programovanie charakterizuje vetvu, ktorá sa zaoberá analýzou, návrhom a implementáciou distribuovaných aplikácií, teda aplikácií, ktoré podporujú paralelizáciu programových činností na množine (spravidla geograficky rozptýlených) počítačov, tvoriacich počítačové siete rôznej topológie. Paralelné a distribuované programovanie spoločne patria do tzv. súbežného, resp. konkurenčného programovania.

Iste, paralelizáciu úloh sme mohli realizovať aj v programovacích jazykoch štruktúrovaného programovania, no v tejto práci sa sústredíme na novú programovaciu paradigmu, ktorú nazývame paralelné objektovo orientované programovanie (POOP). Posun od objektovo orientovaného programovania k paralelnému objektovo orientovanému programovaniu (OOP → POOP) je podľa nás jedinou možnou cestou, ako čeliť výzvam budúcnosti. Už dnes sa dá s veľkou spoľahlivosťou predpovedať posun od architektúry „multi-core“ (procesory s menej ako 8 exekučnými jadrami) k architektúre „many-core“ (procesory s 8 a viac exekučnými jadrami). Je preto evidentné, že ak budeme chcieť, aby naše počítačové aplikácie využili konkurenčnú výhodu v podobe množiacich sa exekučných jadier moderných procesorov, musíme ich navrhnúť s dôrazom na paralelizáciu programových činností. Rovnako rozumné sa javí upotrebiť všetky silné vlastnosti

objektovo orientovaného programovania, ako najúspešnejšieho programovacieho modelu posledných rokov.

8 Kvantifikácia nárastu výkonnosti pri POOP

Komerčných programátorov zaujíma predovšetkým nárast výkonnosti, ktorý dosiahne pôvodne sekvenčná aplikácia po svojej paralelizácii. Empiricky by sme mohli inferovať matematický vzťah pre kvantifikáciu nárastu výkonnosti takto:

$$N_V = \frac{T_S}{T_P(n)}$$

kde:

- N_V je koeficient nárastu výkonnosti aplikácie.
- T_S je najlepší exekučný čas sekvenčnej aplikácie.
- T_P je najlepší exekučný čas paralelnej aplikácie pri použití n programových vlákien a n exekučných jadier viacjadrového procesora (alebo n samostatných procesorov).

Nárast výkonu nám vraví, koľkokrát rýchlejšie bude aplikácia spracovaná vtedy, keď bude paralelizovaná. Uvažujme napríklad konzolovú aplikáciu, ktorá realizuje inicializáciu štvorcovej matice typu 10000x10000 pseudonáhodnými celými číslami z vopred stanoveného intervalu. Ďalej predpokladajme, že čas spracovania sekvenčnej konzolovej aplikácie je 10 sekúnd, zatiaľ čo čas spracovania paralelnej konzolovej aplikácie je 4 sekundy. Podľa uvedeného vzťahu platí:

$$N_V = \frac{T_S}{T_P(n)} = \frac{10}{4(2)} = 2{,}5$$

Ako sme empiricky zistili, paralelná aplikácia je 2,5-krát výkonnejšia ako ekvivalentná sekvenčná aplikácia.

V predchádzajúcom príklade sekvenčná aj paralelná aplikácia implementovala práve jeden algoritmus: algoritmus, ktorý inicializoval maticu pseudonáhodnými celočíselnými hodnotami. Aj keď v ukážke sme použili aplikáciu, ktorej výkonná časť bola tvorená len jedným algoritmom, komerčné aplikácie obsahujú spravidla stovky až tisíce rozmanitých algoritmov, ktoré riešia rozličné úlohy. Ak budeme chcieť modelovať nárast výkonnosti sekvenčnej aplikácie po jej paralelizácii v tomto ponímaní, budeme musieť zaviesť účinnú klasifikáciu algoritmov:

- Nech **A** je množina všetkých algoritmov aplikácie: $A = \{a_1, a_2, \dots, a_n\}$.

- Nech **S** je podmnožina množiny **A**, ktorá obsahuje len sekvenčné algoritmy aplikácie: $S \subset A; S = \{s_1, s_2, \dots, s_n\}$.

- Nech **P** je podmnožina množiny **A**, ktorá obsahuje len paralelné algoritmy aplikácie: $P \subset A; P = \{p_1, p_2, \dots, p_n\}$.

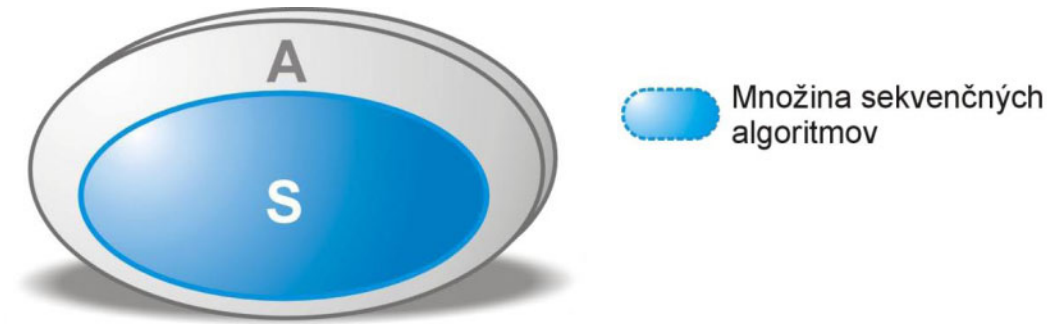
Z pohľadu paralelizácie aplikácie je pre nás významná absolútna početnosť paralelných algoritmov, ktoré uchováva množina **P**. Sú to práve tieto algoritmy, ktoré umožňujú paralelizáciu programových činností. Ak činnosti implementované algoritmami z množiny **P** budú vykonávané súbežne, paralelná časť aplikácie zaznamená citeľný nárast výkonnosti pri svojej exekúcii na počítači s viacjadrovým procesorom. Na druhej strane, algoritmy situované v množine **S** sú rýdzo sekvenčné, čo znamená, že programové činnosti implementované týmito algoritmami nie je možné vykonávať súbežne. Algoritmy množiny **S** budú spracúvané sekvenčne, teda jeden za druhým, s nulovou možnosťou svojej paralelizácie.

V záujme exaktnej analýzy nárastu výkonnosti, ktorý prináša paralelizácia, je nutné stanoviť relatívnu početnosť (R_p) sekvenčných a paralelných algoritmov v počítačovej aplikácii. Podľa finálnej algoritmickej kompozície aplikácie môžeme identifikovať nasledujúce kompozičné modely:

1. **kompozičný model: Všetky algoritmy aplikácie sú sekvenčné**, teda $A = S$. Keď je aplikácia tvorená sekvenčnými algoritmami, ktoré nemožno

paralelizovať, nemôžeme očakávať žiaden nárast výkonnosti pri jej exekúcii na strojoch s viacjadrovými procesormi.

- **Vizualizácia modelu:**

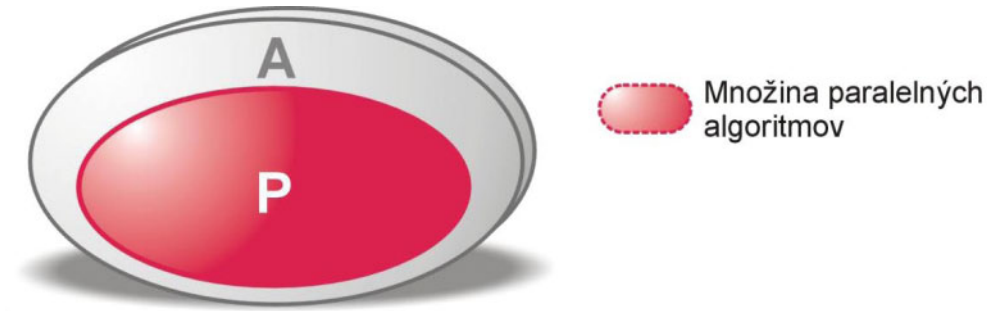


Obr. 8.1: 1. kompozičný model

- **Možnosť zvýšenia výkonnosti pri exekúcii na počítači s viacjadrovým procesorom**: nulová.

2. **kompozičný model: Všetky algoritmy aplikácie sú paralelné**, teda $A = P$. Aplikácia v tejto algoritmickej skladbe dokáže maximálne využiť výkonnostný potenciál paralelne orientovaných počítačových systémov. Ak predpokladáme optimálnu škálovateľnosť aplikácie, ide o ideálny stav, kedy sa paralelná aplikácia dokáže prispôsobiť rôzne výkonnej hardvérovej platforme. Nárast výkonnosti vyjadruje lineárnu, sublineárnu, alebo superlineárnu závislosť medzi počtom exekučných jadier viacjadrového procesora a koeficientom nárastu výkonnosti. Napríklad, ak výkonnosť aplikácie na 2-jadrovom procesore je daná koeficientom 1, tak na 6-jadrovom procesore sa bude pri tendencii lineárneho nárastu výkonnosti rovnať hodnote 3.

- **Vizualizácia modelu:**

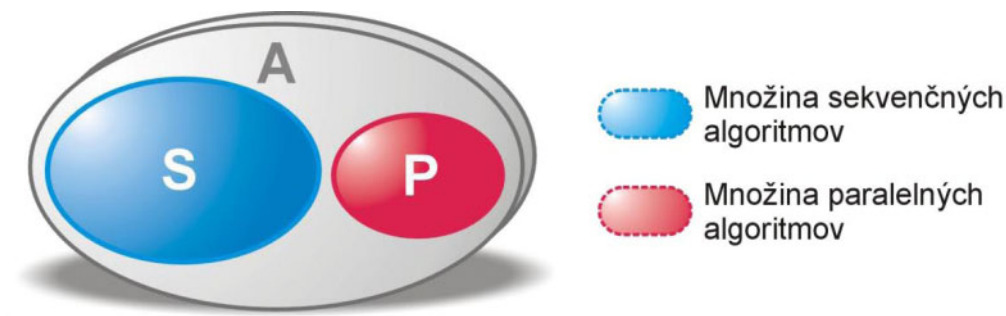


Obr. 8.2: 2. kompozičný model

- **Možnosť zvýšenia výkonnosti pri exekúcii na počítači s viacjadrovým procesorom**: maximálna.

3. **kompozičný model: Aplikácia obsahuje väčšie zastúpenie sekvenčných ako paralelných algoritmov**, teda $A = S + P; Rp(S) > Rp(P)$. Napriek tomu, že relatívna početnosť paralelných algoritmov je menšia ako relatívna početnosť sekvenčných algoritmov, aplikácia zaznamená nárast výkonnosti pri svojej exekúcii na viacjadrových procesoroch. Je však nutné podotknúť, že škálovateľný nárast výkonnosti sa bude dotýkať len paralelných algoritmov. Ak bude aplikácia obsahovať 40 % paralelných algoritmov, spustenie aplikácie na výkonnejšej hardvérovej platforme urýchli spracovanie len tejto množiny algoritmov.

- **Vizualizácia modelu:**

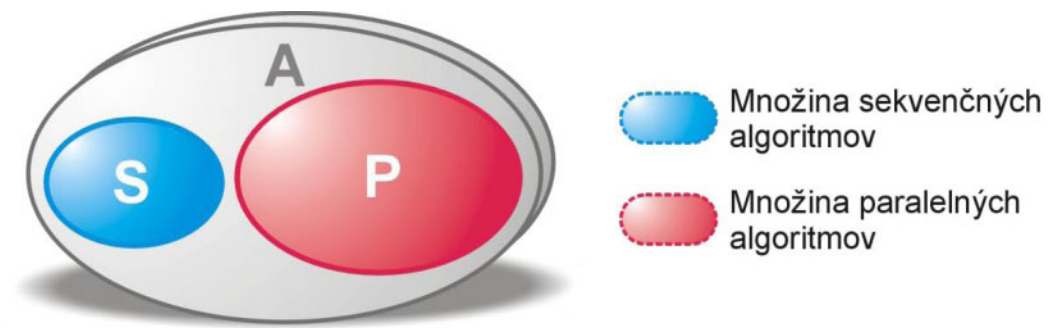


Obr. 8.3: 3. kompozičný model

Možnosť zvýšenia výkonnosti pri exekúcii na počítači s viacjadrovým procesorom: Daná $R_p(P)$.

4. **kompozičný model: Aplikácia obsahuje väčšie zastúpenie paralelných ako sekvenčných algoritmov**, teda $A = S + P; Rp(S) < Rp(P)$. Tento kompozičný model je paralelizácii naklonený viac ako ten predchádzajúci. S narastajúcim zastúpením paralelných algoritmov sa zvyšuje množstvo algoritmov, ktoré možno podrobiť súbežnému spracovaniu. Aplikácia v uvedenom algoritmickom zložení bude disponovať vyšším nárastom výkonnosti v porovnaní s predchádzajúcim typom aplikácie, pretože zvýšenie výkonnosti je limitované menšou relatívnou početnosťou sekvenčných algoritmov.

   - **Vizualizácia modelu:**

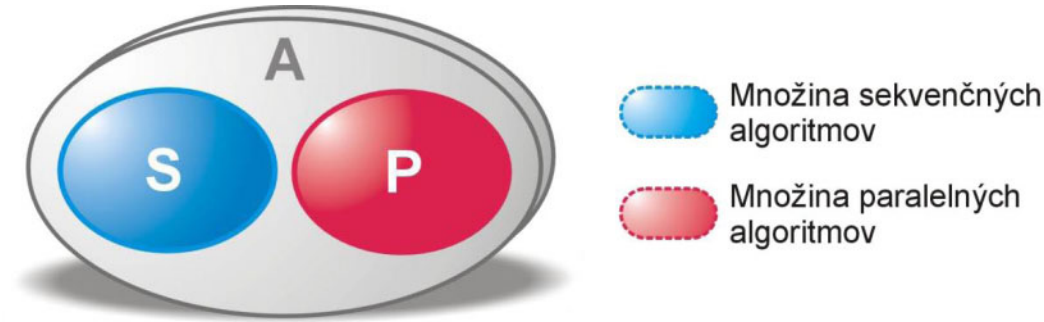


Obr. 8.4: 4. kompozičný model

   - **Možnosť zvýšenia výkonnosti pri exekúcii na počítači s viacjadrovým procesorom**: Daná $R_p(P)$.

5. **kompozičný model: Zastúpenie sekvenčných a paralelných algoritmov v aplikácii je rovnaké**, teda $A = S + P; Rp(P) = Rp(S)$. Identická relatívna početnosť sekvenčných a paralelných algoritmov predstavuje 50 % potenciál zvýšenia výkonnosti počítačovej aplikácie pri jej exekúcii na počítači s viacjadrovým procesorom.

- **Vizualizácia modelu:**



Obr. 8.5: 5. kompozičný model

- **Možnosť zvýšenia výkonnosti pri exekúcii na počítači s viacjadrovým procesorom**: Daná $R_p(P)$.

9 Amdahlov zákon

Americký počítačový vedec Gene Amdahl stanovil v roku 1967 matematický vzťah na určenie maximálneho teoretického nárastu výkonnosti pôvodne sekvenčnej aplikácie po jej paralelizácii. Tento vzťah je v teoretickej informatike známy ako Amdahlov zákon:

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

kde:

- N_V je koeficient maximálneho možného nárastu výkonnosti aplikácie.
- T_S je exekučný čas spracovania sekvenčných algoritmov aplikácie.
- T_P je exekučný čas spracovania paralelných algoritmov aplikácie.
- n je počet exekučných jadier procesora, resp. počet samostatných procesorov.

Príklad: Uvažujme sekvenčnú aplikáciu, ktorá implementuje 100 algoritmov. Jeden algoritmus vykonáva v priemere 50 elementárnych operácií (ηOs), pričom spracovanie 1 elementárnej operácie trvá 1 milisekundu (ms). Čas výkonu jedného algoritmu bude 50 ms a celkový čas na realizáciu sekvenčnej aplikácie bude 5000 ms. Po analýze algoritmickej štruktúry sekvenčnej aplikácie zistíme, že 60 % algoritmov môžeme paralelizovať. Zvyšných 40 % algoritmov je rýdzo sekvenčných, a teda nepripúšťajú žiadnu možnosť paralelného spracovania uskutočňovaných programových činností. Na základe Amdahlovho zákona zisťujeme tieto skutočnosti:

1. Maximálny teoretický nárast výkonnosti aplikácie spustenej na jednojadrovom procesore:

$$N_V = \frac{T_S + T_P}{T_S + \frac{T_P}{n}} = \frac{2000 + 3000}{2000 + \frac{3000}{1}} = \frac{5000}{5000} = 1{,}000$$

2. Maximálny teoretický nárast výkonnosti aplikácie spustenej na 2-jadrovom procesore:

$$N_V = \frac{T_S + T_P}{T_S + \frac{T_P}{n}} = \frac{2000 + 3000}{2000 + \frac{3000}{2}} = \frac{5000}{2000 + 1500} = \frac{5000}{3500} = 1{,}428$$

3. Maximálny teoretický nárast výkonnosti aplikácie spustenej na 4-jadrovom procesore:

$$N_V = \frac{T_S + T_P}{T_S + \frac{T_P}{n}} = \frac{2000 + 3000}{2000 + \frac{3000}{4}} = \frac{5000}{2000 + 750} = \frac{5000}{2750} = 1{,}818$$

Ako vidíme, v prvom prípade nedosahuje aplikácia žiaden nárast výkonu. To je však pochopiteľné, pretože aplikácia je spustená na jednojadrovom procesore, ktorý neumožňuje paralelnú exekúciu programových inštrukcií. Navzdory faktu, že 60 % algoritmov aplikácie je paralelných, tieto budú spracúvané pseudoparalelne. Zvyšných 40 % sekvenčných algoritmov bude realizovaných sekvenčne.

V ďalších dvoch prípadoch kvantifikujeme pomocou Amdahlovho zákona nárast výkonnosti aplikácie. Je zrejmé, že so zvyšujúcou sa výpočtovou kapacitou hardvérovej platformy rastie aj koeficient zvýšenia výkonnosti počítačovej aplikácie.

V súvislosti s Amdahlovým zákonom si dovolíme poukázať na interesantné praktické implikácie:

1. **Otázka preferencie**: *Má väčší význam zvyšovať výkon hardvérovej infraštruktúry, teda počet exekučných jadier procesora, alebo zvyšovať zastúpenie paralelných algoritmov v algoritmickej skladbe počítačovej aplikácie?*

   Vychádzajme z našich doterajších úvah o imaginárnej počítačovej aplikácii. Vypočítali sme, že ak bude 60 % všetkých algoritmov spracúvaných

paralelne, tak na počítači s 2-jadrovým procesorom môžeme v ideálnom prípade počítať s takmer 43% nárastom výkonnosti. Aký nárast výkonnosti však na rovnakom počítači zaznamenáme, ak po prepracovaní aplikácie zistíme, že sme schopní zvýšiť zastúpenie paralelných algoritmov z 60 % na 80 %? V tomto prípade bude sekvenčných len 20 zo 100 algoritmov. Podľa Amdahlovho zákona platí:

$$N_V = \frac{T_S + T_P}{T_S + \frac{T_P}{n}} = \frac{1000 + 4000}{1000 + \frac{4000}{2}} = \frac{5000}{1000 + 2000} = \frac{5000}{3000} = 1{,}666$$

Výkonnosť aplikácie v zložení $\frac{20}{80}$ $(R_p(S)/R_p(P))$ bude vyššia ako pôvodne, a to na rovnako výkonnej počítačovej stanici. Preto smieme konštatovať, že väčšiu preferenciu má zvyšovanie relatívnej početnosti paralelných algoritmov počítačovej aplikácie. Prirodzene, ešte väčší nárast výkonnosti aplikácie nameriame, ak ju v zložení $\frac{20}{80}$ spustíme na výkonnejšom stroji. Pre exekúciu aplikácie na počítači so 4-jadrovým procesorom platí:

$$N_V = \frac{T_S + T_P}{T_S + \frac{T_P}{n}} = \frac{1000 + 4000}{1000 + \frac{4000}{4}} = \frac{5000}{1000 + 1000} = \frac{5000}{2000} = 2{,}5$$

Aplikácia s $\frac{20}{80}$-algoritmickou skladbou bude na 4-jadrovom procesore spracúvaná 2,5-krát rýchlejšie.

2. **Otázka maximalizácie efektivity**: *Aký maximálny úžitok môžeme získať paralelizáciou počítačovej aplikácie?*

   Vychádzajúc z Amdahlovho zákona môžeme vyhlásiť, že exekučný čas potrebný na spracovanie paralelných algoritmov aplikácie, bude klesať v závislosti od vzrastajúceho počtu exekučných jadier procesora. Ak budeme paralelnú verziu aplikácie porovnávať s najvýkonnejšou sekvenčnou verziou tejto aplikácie, tak smieme Amdahlov zákon upraviť takto:

$$N_V = \frac{T(1)}{T_S + \frac{T_P}{n}}$$

Do čitateľa zlomku sme umiestnili člen $T(1)$. Tento člen predstavuje najlepší exekučný čas, aký pri svojom spracovaní zaznamenala sekvenčná aplikácia. Venujme sa teraz menovateľu zlomku. Ak sa bude počet exekučných jadier procesora (n) zväčšovať, tak hodnota výrazu $\frac{T_P}{n}$ sa bude zmenšovať. Teoreticky, pre $n \to \infty$ sa hodnota výrazu $\frac{T_P}{n}$ bude limitne blížiť k nule, teda $\lim_{n\to\infty} \frac{T_P}{n} = 0$. Z uvedeného plynie, že počnúc určitým hraničným bodom nebude zmysluplné pokračovať vo zvyšovaní počtu exekučných jadier procesora, pretože dodatočný nárast výpočtovej kapacity stroja nebude mať signifikantný vplyv na ďalšie zvýšenie výkonnosti počítačovej aplikácie. Elimináciou výrazu $\frac{T_P}{n}$ dostávame nasledujúcu deriváciu Amdahlovho zákona:

$$N_V = \frac{T(1)}{T_S}$$

kde:

- $T(1)$ je najlepší exekučný čas sekvenčnej aplikácie.
- T_S je exekučný čas potrebný na spracovanie sekvenčných algoritmov v paralelizovanej verzii aplikácie.

Praktickou implikáciou tohto matematického vzťahu je skutočnosť, že maximálny možný nárast výkonnosti paralelizovanej počítačovej aplikácie bude vždy limitovaný exekučným časom, ktorého alokáciu si vyžiadajú sekvenčné algoritmy. V prípade našej aplikácie v algoritmickom zložení $\frac{20}{80}$ platí:

$$N_V = \frac{T(1)}{T_S} = \frac{5000}{1000} = 5$$

Paralelizáciou aplikácie získame maximálne 5-násobný nárast jej výkonnosti. V tab. 9.1 ukazujeme, ako sa zvyšuje výkonnosť aplikácie pri zvyšovaní počtu exekučných jadier procesora.

s	**p**	**ηOs**	**T_S**	**T_P**	**n**	**T_P/n**	**N_V**
20	80	50	1000	4000	1	4000	1
20	80	50	1000	4000	2	2000	1,666667
20	80	50	1000	4000	3	1333,333	2,142857
20	80	50	1000	4000	4	1000	2,5
20	80	50	1000	4000	5	800	2,777778
20	80	50	1000	4000	10	400	3,571429
20	80	50	1000	4000	20	200	4,166667
20	80	50	1000	4000	40	100	4,545455
20	80	50	1000	4000	80	50	4,761905
20	80	50	1000	4000	160	25	4,878049
20	80	50	1000	4000	320	12,5	4,938272
20	80	50	1000	4000	640	6,25	4,968944
20	80	50	1000	4000	1280	3,125	4,984424
20	80	50	1000	4000	2560	1,5625	4,9922
20	80	50	1000	4000	5120	0,78125	4,996097
20	80	50	1000	4000	10240	0,390625	4,998048

Tab. 9.1: Kvantifikácia relácie medzi výkonnosťou aplikácie a počtom exekučných jadier procesora

Legenda k tab. 9.1:

- s je relatívna početnosť sekvenčných algoritmov aplikácie.
- p je relatívna početnosť paralelných algoritmov aplikácie.
- ηOs je počet elementárnych operácií pripadajúcich na 1 algoritmus.
- T_S je celkový exekučný čas spracovania sekvenčných algoritmov aplikácie.

- T_P je celkový exekučný čas spracovania paralelných algoritmov aplikácie.
- n je počet exekučných jadier procesora.
- N_V je koeficient nárastu výkonnosti.

Tab. 9.1 ukazuje, ako sa mení nárast výkonnosti aplikácie s algoritmickou skladbou $\frac{20}{80}$ pri zvyšovaní počtu exekučných jadier procesora. Ako vidíme, výkonnosť aplikácie sa dynamicky zvyšuje až do okamihu, keď sa počet exekučných jadier dostane na hodnotu 80. Ak by sme teda aplikáciu spustili na počítači s 80-jadrovým procesorom, teoretický nárast výkonnosti by bol približne 4,76. Po ďalšej analýze zisťujeme, že pokusy o dodatočné zvýšenie výpočtovej kapacity počítača neprinášajú požadovaný efekt v náraste výkonu aplikácie. Ak totiž zdvojnásobíme počet exekučných jadier procesora (na hodnotu 160), celkový nárast výkonnosti bude 4,88, čo znamená, že citeľné rozšírenie výpočtových kapacít stroja generuje len veľmi malý prírastok (0,12) v náraste výkonnosti aplikácie. Ak budeme ďalej zvyšovať počet exekučných jadier procesora, relatívne sa bude nárast výkonnosti aplikácie stále znižovať. Je zrejmé, že teoretická hranica výkonnosti aplikácie bude limitovaná koeficientom 5.

10 Gustafsonov zákon

V roku 1988 americký počítačový vedec John Gustafson vyslovil modifikáciu Amdahlovho zákona, ktorá je v teoretickej informatike známa ako Gustafsonov zákon[21]. Gustafsonov zákon poukazuje na potenciálnu neplatnosť dvoch premís, z ktorých Amdahlov zákon vychádza:

1. **premisa: Veľkosť riešeného problému má fixnú povahu.** Amdahlov zákon predpokladá, že veľkosť riešeného problému (resp. veľkosť inštancie riešeného problému) sa so vzrastajúcou výpočtovou kapacitou počítačovej

[21] Gustafsonov zákon sa niekedy nazýva aj Gustafsonov-Barsisov zákon.

stanice nemení. Napriek tomu, že veľkosť riešeného problému môže zostať za istých podmienok invariantná voči zvyšujúcemu sa počtu exekučných jadier procesora, resp. voči zvyšujúcemu sa počtu samostatných procesorov viacprocesorového stroja, praktické experimenty dokazujú, že vo väčšine prípadov táto premisa neplatí. Vo všeobecnosti, ak budeme pracovať s rádovo výkonnejším počítačovým systémom, tak budeme chcieť za rovnako dlhý exekučný čas spracovať problém s väčšou zložitosťou. Vzhľadom na tendenciu nárastu veľkosti riešeného problému priamo úmerne s nárastom výpočtovej kapacity počítača možno vyhlásiť, že nie veľkosť problému, ale exekučný čas algoritmov potrebných na vyriešenie tohto problému, je fixnou kategóriou.

2. **premisa: Sekvenčný algoritmus je najlepším riešením problému.** Táto premisa vyplýva zo skutočnosti, že výkonnosť paralelizovanej aplikácie je porovnávaná s výkonnosťou najlepšej sekvenčnej aplikácie. Určitú množinu problémov však môžeme efektívnejšie vyriešiť pomocou paralelných a nie sekvenčných algoritmov. Pritom počet krokov paralelných algoritmov v celkovom výpočtovom procese smie byť oveľa menší v porovnaní s kompozíciou sekvenčných algoritmov.

Gustafsonov zákon vraví, že medzi veľkosťou riešeného problému a výpočtovou kapacitou stroja (meranou v našom ponímaní počtom exekučných jadier procesora) existuje lineárna závislosť. Ak znásobíme počet exekučných jadier procesora, získame stroj s väčšou výpočtovou kapacitou. Veľkosť problému bude mať tendenciu prispôsobiť sa dodatočnému výkonu hardvérovej platformy.

Gustafsonov zákon môžeme formalizovať nasledujúcim spôsobom:

$$N_V = \frac{T_S + T_P \times n}{T_S + T_P}$$

kde:

- N_V je škálovateľný nárast výkonnosti počítačovej aplikácie.
- T_S je exekučný čas potrebný na spracovanie sekvenčných algoritmov.
- T_P je exekučný čas potrebný na spracovanie paralelných algoritmov.
- n je počet exekučných jadier viacjadrového procesora, resp. počet procesorov viacprocesorového stroja.

Príklad: Predpokladajme algoritmické zloženie $\frac{40}{60}$ $(Rp(S)/Rp(P))$ počítačovej aplikácie. Každý algoritmus nech realizuje 50 elementárnych operácií (ηOs), pričom uskutočnenie 1 ηOs trvá 1ms. Aký bude škálovateľný nárast výkonu počítačovej aplikácie?

$$N_V = \frac{T_S + T_P \times n}{T_S + T_P} = \frac{2000 + 3000 \times 2}{2000 + 3000} = \frac{8000}{5000} = 1{,}6$$

Rovnako môžeme upotrebiť aj upravenú podobu Gustafsonovho zákona:

$$N_V = n + (1 - n) \times s$$

kde:

- N_V je škálovateľný nárast výkonnosti počítačovej aplikácie.
- n je počet exekučných jadier viacjadrového procesora, resp. počet procesorov viacprocesorového stroja.
- s je pomer exekučného času sekvenčných algoritmov a celkového exekučného času, teda:

$$s = \frac{T_S}{T_S + T_P}$$

Po úprave získavame:

$$N_V = n + (1 - n) \times \frac{T_S}{T_S + T_P}$$

Pre náš príklad platí:

$$N_V = n + (1 - n) \times \frac{T_S}{T_S + T_P} = 2 + (1 - 2) \times \frac{2000}{2000 + 3000} =$$
$$= 2 + (-1) \times \frac{2000}{5000} = 2 + (-1) \times 0{,}4 = 1{,}6$$

Výsledkom je 60% nárast výkonnosti počítačovej aplikácie.

10.1 Komparácia Amdahlovho a Gustafsonovho zákona

Amdahlov zákon vraví, že výkonnosť paralelizovanej aplikácie pri riešení problému fixnej veľkosti bude so zvyšujúcou sa výpočtovou kapacitou počítačového systému narastať až pokiaľ nedosiahne určitý hraničný bod. Tento hraničný bod stanovuje maximálny teoretický nárast výkonnosti, čo znamená, že dodatočné zvýšenie výkonu hardvérovej platformy už neprinesie žiaden dodatočný výkonnostný efekt. Gustafsonov zákon vraví, že škálovateľnosť počítačovej aplikácie je závislá od zvýšenia výkonnostnej kapacity počítačového systému. Keďže veľkosť riešeného problému nie je fixná kategória, podľa Gustafsona je možné vždy zaznamenať nárast výkonnosti aplikácie, a to za predpokladu, ak bude veľkosť riešeného problému narastať spoločne s výkonom počítačovej stanice. Kým podľa Amdahlových premís dochádza so zvyšovaním výpočtovej kapacity systému k znižovaniu exekučného času výkonu algoritmov počítačovej aplikácie, Gustafson stanovuje, že škálovateľná aplikácia bude schopná za rovnako dlhý exekučný čas spracovať väčšie množstvo práce. Ak vychádzame z Gustafsonovho zákona, môžeme konštatovať nasledujúce: Pokiaľ je aplikácia škálovateľná, tak keď znásobíme veľkosť problému (množstvo vykonanej práce) a rovnako lineárne zvýšime výkonnosť stroja (počet exekučných jadier procesora, resp. počet procesorov), získame adekvátny nárast výkonnosti paralelizovanej počítačovej aplikácie.

11 Lineárny, sublineárny a superlineárny nárast výkonnosti pri POOP

Pri analýze paralelných programov sme schopní diagnostikovať tri základné typy nárastu ich výkonnosti:

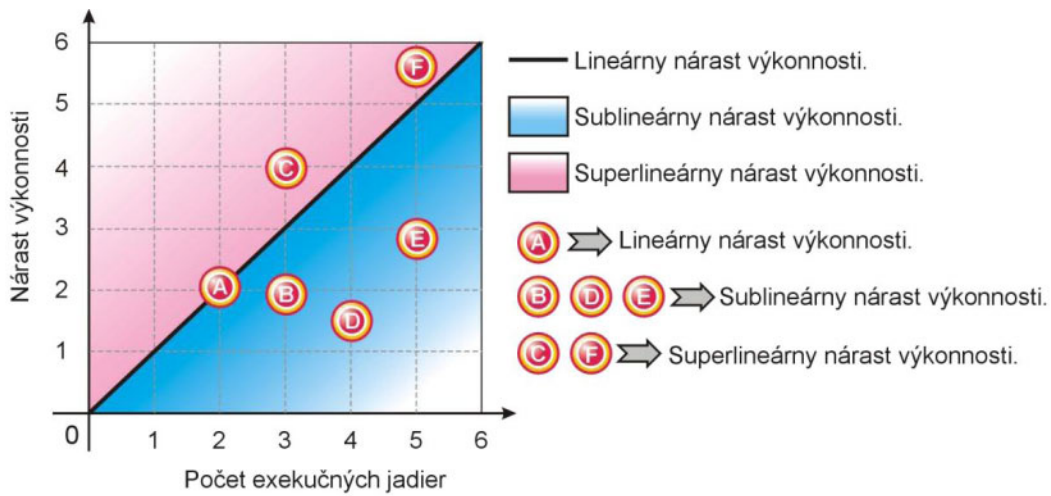
1. **Lineárny nárast výkonnosti.** Ak platí rovnica $N_V = n$, kde N_V je nárast výkonnosti programu po jeho paralelizácii a n je počet exekučných jadier viacjadrového procesora, tak vravíme, že program preukazuje lineárny nárast výkonnosti. Tento typ nárastu výkonnosti programu je vo väčšine prípadov ideálnym stavom, teda stavom, kedy každé dodatočné zvýšenie výpočtových kapacít počítačovej stanice spôsobí adekvátne zvýšenie výkonnosti programu. Keď ako mieru zvyšovania výkonnosti programu použijeme skracovanie jeho exekučného času, tak pri lineárnej výkonnostnej profilácii sa exekučný čas programu skráti n-krát vždy, ak sa n-krát zvýši počet exekučných jadier viacjadrového procesora.

2. **Sublineárny nárast výkonnosti.** Hoci teoreticky je paralelný program schopný dosiahnuť lineárny nárast výkonnosti, väčšina paralelných programov spracúvaných v praktických podmienkach vykazuje menší ako lineárny nárast svojej výkonnosti. Ak platí rovnica $N_V < n$ kde N_V je nárast výkonnosti programu po jeho paralelizácii a n je počet exekučných jadier viacjadrového procesora, tak vravíme, že výkonnosť programu je sublineárna. Sublineárne správanie sa výkonnostnej funkcie spôsobujú najmä výkonnostné penalizácie, ktoré súvisia s vedľajšími nákladmi generovanými riadením životných cyklov programových vlákien, ich synchronizáciou a suboptimálnym využívaním dostupných vyrovnávacích pamätí.

3. **Superlineárny nárast výkonnosti.** V ojedinelých prípadoch môžeme pri skúmaní paralelných programov detegovať vyšší ako lineárny nárast výkonnosti. Za týchto okolností platí, že $N_V > n$, kde N_V je nárast výkonnosti programu po jeho paralelizácii a n je počet exekučných jadier viacjadrového

procesora. To znamená, že n-násobné zvýšenie počtu exekučných jadier viacjadrového procesora spôsobí vyšší ako n-násobný nárast výkonnosti paralelného programu. Tento extremálny stav súvisí predovšetkým s optimálnym využívaním vyrovnávacích pamätí, tzv. cache-efektom. Ak sú dáta, s ktorými algoritmy paralelného programu manipulujú, kompletne načítané do vyrovnávacích pamätí, dochádza k eliminácii výkonnostných penalizácií, ktoré by inak vznikali v súvislosti s pamäťovou latenciou zapríčinenou lokalitou dát.

Vizualizácia lineárneho, sublineárneho a superlineárneho nárastu výkonnosti paralelného programu je znázornená na obr. 11.1.



Obr. 11.1: Vizualizácia lineárneho, sublineárneho a superlineárneho nárastu výkonnosti paralelného programu

Komentár k obr. 11.1: V schéme sledujeme závislosť medzi počtom exekučných jadier počítačovej stanice a nárastom výkonnosti paralelných programov označených identifikátormi **A** až **F**. Program **A** disponuje lineárnym nárastom výkonnosti, pretože pri svojej aktivácii na 2-jadrovom procesore zaznamená 2-násobný nárast výkonnosti. Programy **B**, **D** a **E** sú umiestnené v modrej oblasti, ktorá predstavuje sublineárny nárast výkonnosti. Z uvedených programov je najmenej

výkonný program **D**, pretože ak ho spustíme na počítači so 4-jadrovým procesorom, jeho výkonnosť nie je ani dvojnásobná. Programy **C** a **F** sú zakreslené v ružovej oblasti, ktorá združuje paralelné programy so superlineárnym nárastom výkonnosti. V skutočnosti je program **C** spustený na 3-jadrovom procesore 4-krát výkonnejší, ako keby sme ho spustili na počítači s jednojadrovým procesorom. Program **F** zaznamenáva na počítači s 5 jadrami takmer 6-násobný nárast svojej výkonnosti.

12 Konštrukcia programových vlákien v jazyku C# 3.0

Vývojári pracujúci s programovacím jazykom C# 3.0 majú viacero možností, ako pristupovať k tvorbe paralelných aplikácií. V tejto kapitole rozoberieme explicitný paralelizmus, ktorý spočíva v manuálnom vytváraní programových vlákien a v priamom injektovaní zdrojového kódu, ktorý bude na týchto vláknach spracúvaný.

12.1 Manipulácia s primárnym programovým vláknom

Na vyššej úrovni abstrakcie je programové vlákno v jazyku C# 3.0 reprezentované inštanciou triedy **Thread** z menného priestoru **System.Threading**.

Nasledujúci fragment zdrojového kódu jazyka C# 3.0 ukazuje, ako získame základné informácie o primárnom programovom vlákne jednovláknovej riadenej aplikácie:

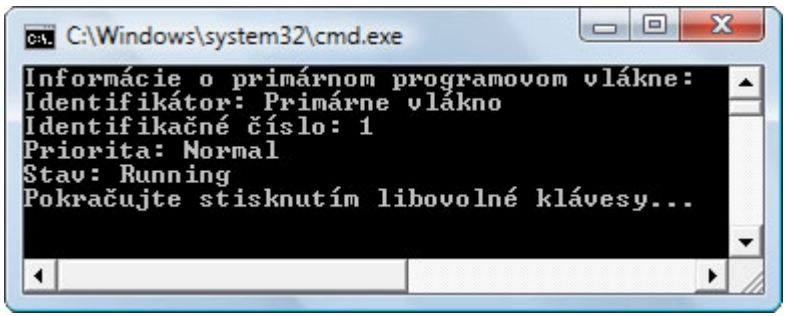
```
static void Main(string[] args)
{
    // Získanie odkazu na primárne programové vlákno.
    Thread primárneVlákno = Thread.CurrentThread;
    // Nastavenie identifikátora primárneho programového vlákna.
    primárneVlákno.Name = "Primárne vlákno";
    Console.WriteLine("Informácie o primárnom programovom vlákne:" +
        "\nIdentifikátor: " + primárneVlákno.Name +
        "\nIdentifikačné číslo: " + primárneVlákno.ManagedThreadId +
        "\nPriorita: " + primárneVlákno.Priority +
        "\nStav: " + primárneVlákno.ThreadState);
}
```

Ak nie je stanovené inak, je každá novo založená riadená aplikácia jednovláknová. Keďže primárne programové vlákno vytvára automaticky virtuálny exekučný systém, môžeme s ním okamžite pracovať. Odkaz na objekt (inštanciu triedy **Thread**), ktorý reprezentuje primárne programové vlákno, získame volaním skalárnej statickej vlastnosti **CurrentThread** triedy **Thread** (táto vlastnosť je určená len na čítanie). Odkaz na primárne programové vlákno ukladáme do typovo silnej odkazovej premennej s identifikátorom **primárneVlákno**.

Hoci virtuálny exekučný systém disponuje kompetenciami na vytvorenie primárneho programového vlákna, toto vlákno nie je implicitne pomenované. To znamená, že primárne programové vlákno nemá svoj identifikátor. Vo všeobecnosti platí konvencia, aby vývojári jednotlivé vlákna náležite pomenovali, pretože ak budú všetky vlákna disponovať svojimi identifikátormi, ich monitorovanie (a to najmä pri ladení aplikácie) bude oveľa jednoduchšie.

Primárnemu programovému vláknu našej riadenej aplikácie prisudzujeme identifikátor pomocou skalárnej inštančnej vlastnosti **Name**. Identifikátorom vlákna sa môže stať akákoľvek korektne zapísaná postupnosť textových znakov a cifier, ktorá vyhovuje nomenklatúrnym pravidlám pre pomenovanie programových entít jazyka C# 3.0.

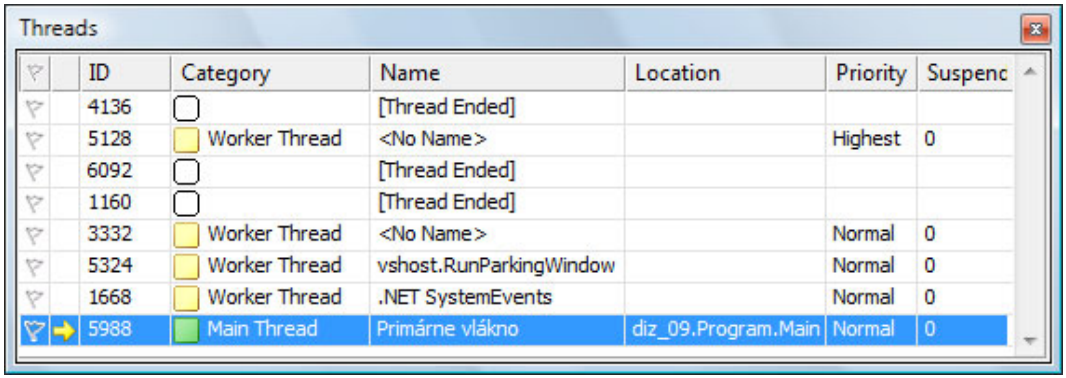
Každé vlákno charakterizuje množina metadát, ku ktorým patrí okrem identifikátora identifikačné číslo (jednoznačný identifikátor vlákna medzi ostatnými vláknami), priorita vlákna a aktuálny stav vlákna. Po spustení uvedeného fragmentu zdrojového kódu zistíme, že naše primárne programové vlákno sa volá „Primárne vlákno“, jeho identifikačné číslo je 1, má normálnu prioritu a je práve spustené (stav „Running“).



Obr. 12.1: Základné údaje o primárnom programovom vlákne

Ak na riadok kódu, v ktorom dochádza k volaniu metódy **WriteLine** triedy **Console** umiestnime lokálny bod prerušenia (klávesom F9, resp. kliknutím do sivého vertikálneho pruhu) a aplikáciu spustíme, môžeme prostredníctvom dialógového okna **Threads** monitorovať zloženie jedno- alebo viacvláknových riadených aplikácií (obr. 12.2).

V dialógovom okne **Threads** sa nachádza viacero stĺpcov, ktoré charakterizujú jednotlivé programové vlákna. Keďže pri ladení riadených aplikácií vytvára ladiaci program (debugger) viacero pomocných pracovných vlákien, vidíme v okne **Threads** väčší počet vlákien. Primárne programové vlákno je však práve jedno, pričom v stĺpci **Category** je identifikované ako **Main Thread**. V stĺpci **ID** je uvedené interné identifikačné číslo primárneho programového vlákna (interné identifikačné číslo primárneho programového vlákna je iné ako identifikačné číslo tohto vlákna, ktoré sa zobrazuje na výstupe).



Obr. 12.2: Dialógové okno **Threads**

Každé programové vlákno je asociované s istou metódou, zdrojový kód ktorej je na príslušnom vlákne vykonávaný. V prípade primárneho programového vlákna môžeme túto asociáciu badať v stĺpci **Location**. Vidíme, že na primárnom programovom vlákne je spracúvaný kód hlavnej metódy **Main** (to je fixný, a teda nemenný stav). V predposlednom stĺpci **Priority** je zobrazená priorita programového vlákna. Primárne programové vlákno má normálnu prioritu. Normálny stupeň priority majú okrem primárneho programového vlákna aj všetky programátorom vytvorené pracovné vlákna, ak nie je stanovené inak. Vlákno, ktorého zdrojový kód je práve podrobený exekúcii, je označené žltou šípkou (druhý stĺpec zľava).

12.2 Manipulácia s pracovným programovým vláknom

Vývojári môžu vytvárať svoje vlastné pracovné programové vlákna. To sa na úrovni zdrojového kódu jazyka C# 3.0 deje inštanciovaním triedy **Thread**:

```
class Program
{
    static void Main(string[] args)
    {
        // Vytvorenie nového pracovného vlákna.
        Thread pracovnéVlákno = new Thread(MetódaPracovnéhoVlákna);
        // Pomenovanie pracovného vlákna.
```

```
            pracovnéVlákno.Name = "Pracovné vlákno 1";
            // Spustenie exekúcie na pracovnom vlákne.
            pracovnéVlákno.Start();
            // Čakanie, pokým pracovné vlákno nedokončí svoju činnosť.
            pracovnéVlákno.Join();
        }

        // Definícia metódy, ktorá bude spracúvaná na pracovnom vlákne.
        private static void MetódaPracovnéhoVlákna()
        {
            Console.WriteLine("Prebieha výpočet faktoriálu...");
            Stopwatch sw = new Stopwatch();
            sw.Start();
            long faktoriál = 1;
            for (int i = 1; i <= 20; i++)
            {
                faktoriál *= i;
            }
            sw.Stop();
            Console.WriteLine("Úloha je splnená.");
            Console.WriteLine("20! je {0}.", faktoriál);
            Console.WriteLine("Exekučný čas: {0} ms.",
                sw.Elapsed.TotalMilliseconds);
        }
}
```

Komentár k zdrojovému kódu: Každé pracovné vlákno musí byť asociované s metódou, ktorá bude na tomto vlákne podrobená exekúcii. Pri pohľade na inštanciačný príkaz triedy **Thread** vidíme, že konštruktoru odovzdávame cieľovú metódu. V skutočnosti je tento proces komplikovanejší, pretože konštruktoru neodovzdávame cieľovú metódu, ale odkaz na inštanciu delegáta **ThreadStart**, ktorá v sebe zapuzdruje odkaz na cieľovú metódu.

Inštanciačný príkaz v znení

```
Thread pracovnéVlákno = new Thread(MetódaPracovnéhoVlákna);
```

nedáva explicitne najavo, že dochádza aj ku skonštruovaniu delegáta **ThreadStart**. V záujme lepšej názornosti preto smieme pôvodný inštanciačný príkaz upraviť nasledujúcim spôsobom:

```
Thread pracovnéVlákno = new Thread(
    new ThreadStart(MetódaPracovnéhoVlákna));
```

Prirodzene, v takto zapísanom zdrojovom kóde dokážeme okamžite rozoznať inštanciáciu delegáta, ktorému poskytujeme odkaz na cieľovú metódu so stanoveným identifikátorom.

Po spracovaní inštanciačného príkazu triedy **Thread** existuje nové pracovné vlákno. Toto pracovné vlákno je reprezentované objektom, ktorý sa nachádza v riadenej halde. Je dôležité poznamenať, že vytvorenie pracovného vlákna neimplikuje jeho okamžité spustenie. Spustenie pracovného vlákna je operácia, ktorá znamená asynchrónne vykonanie kódu cieľovej metódy na pracovnom vlákne. Vytvorené pracovné vlákno spustíme volaním metódy **Start** inštancie triedy **Thread**.

Aby sme pracovné vlákno ľahko rozoznali od primárneho vlákna (a dodatočných vlákien, ktoré zhotovuje ladiaci program), prisudzujeme mu používateľsky prívetivý identifikátor.

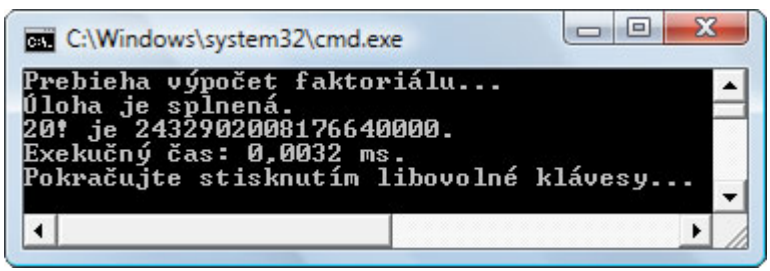
Cieľová metóda, ktorá je spojená s pracovným vláknom, je v našej praktickej ukážke súkromná statická metóda, ktorá je definovaná v primárnej triede programu. Metóda je bezparametrická a nevracia ani žiadnu návratovú hodnotu. Tieto charakteristiky metódy vyplývajúce z jej signatúry sú dané požiadavkami na zhodu medzi signatúrou metódy a signatúrou delegáta, s inštanciou ktorého je metóda previazaná. Ako vysvetlíme o chvíľu, konštruktor triedy **Thread** je preťažený a dovoľuje nám pracovať aj s parametrickými cieľovými metódami a parametrickými delegátmi.

V tele cieľovej metódy dochádza k výpočtu faktoriálu čísla 20. Výpočet faktoriálu je realizovaný cyklom, pričom metóda meria čas, ktorý je potrebný na úspešné dokončenie výpočtového procesu.

Po spustení programu sa najskôr začne vykonávať zdrojový kód hlavnej metódy **Main** (to je pochopiteľné, pretože hlavná metóda je spracúvaná na primárnom programovom vlákne). Po zhotovení pracovného vlákna, vytvorení inštancie

delegáta a nadviazaní spojenia medzi inštanciou delegáta a cieľovou statickou metódou dochádza k spusteniu pracovného vlákna. Po spustení pracovného vlákna je kód cieľovej metódy injektovaný do tohto vlákna a vzápätí okamžite exekvovaný. Ak bude program spustený na počítači s jednojadrovým procesorom, jeho exekúcia bude pseudoparalelná (existujú 2 programové vlákna, ktorých exekúcia bude realizovaná zdieľaním výpočtových prostriedkov procesora pomocou časových kvánt). V prípade, že program spustíme na 2-jadrovom procesore, operačný systém zabezpečí distribúciu vlákien na obe jadrá procesora. Tým zabezpečí ich paralelnú exekúciu.

Naša demonštrácia je triviálna, lebo primárne vlákno nevykonáva súbežne žiadnu inú činnosť. Len čo spustíme pracovné vlákno, voláme jeho metódu **Join**, ktorá zabráni postupu primárneho vlákna až do momentu, kedy pracovné vlákno nedokončí svoju činnosť. To sa stane vtedy, keď bude hotové spracovanie metódy, ktorá je realizovaná na pracovnom vlákne. Výstup programu je uvedený na obr. 12.3.



Obr. 12.3: Výstup 2-vláknového programu

Ak budeme chcieť vykonávať na pracovnom vlákne zdrojový kód parametrickej cieľovej metódy, budeme musieť použiť inštanciu delegáta **ParameterizedThreadStart**:

```
class Program
{
    static void Main(string[] args)
    {
        Thread pracovnéVlákno = new Thread(
            new ParameterizedThreadStart(MetódaPracovnéhoVlákna));
        pracovnéVlákno.Name = "Pracovné vlákno 1";
```

```
            pracovnéVlákno.Start(25);
            pracovnéVlákno.Join();
        }
        private static void MetódaPracovnéhoVlákna(object parameter)
        {
            Console.WriteLine("Prebieha výpočet faktoriálu...");
            Stopwatch sw = new Stopwatch();
            sw.Start();
            int n = Convert.ToInt32(parameter);
            long faktoriál = 1;
            for (int i = 1; i <= n; i++)
            {
                faktoriál *= i;
            }
            sw.Stop();
            Console.WriteLine("Úloha je splnená.");
            Console.WriteLine("{0}! je {1}.", n, faktoriál);
            Console.WriteLine("Exekučný čas: {0} ms.",
                sw.Elapsed.TotalMilliseconds);
        }
}
```

Komentár k zdrojovému kódu: Prvá modifikácia sa dotýka inštanciačného príkazu triedy **Thread**, v ktorom vytvárame inštanciu parametrického delegáta **ParameterizedThreadStart**. Inštancia parametrického delegáta bude spojená s rovnako parametrickou cieľovou metódou. Vzhľadom na to, že signatúra parametrického delegáta definuje jeden formálny parameter typu **object**, budeme musieť formálny parameter s identickým dátovým typom použiť aj pri definícii cieľovej metódy.

Deklarácia parametrického delegáta **ParameterizedThreadStart** je takáto:

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

Druhá modifikácia je spojená s definíciou cieľovej metódy. Definovaný formálny parameter slúži na uchovanie člena, ktorého faktoriál budeme chcieť vypočítať. No keďže je formálny parameter typu **object**, musíme uskutočniť explicitnú typovú konverziu na konkrétny dátový typ (v našom prípade je to 32-bitový celočíselný typ **int**).

Tretia zmena ovplyvňuje spôsob, akým je spúšťané pracovné vlákno. Zahájenie spracovania zdrojového kódu na pracovnom vlákne je odštartované volaním metódy **Start**, ktorá je rovnako parametrická. Metóde **Start** ponúkame celočíselnú hodnotu predstavujúcu člen, ktorého faktoriál chceme vypočítať.

Po komplexnej analýze upraveného kódu smieme konštatovať, že parametrické entity, s ktorými sa stretávame, sú tri:

1. Parametrický delegát **ParameterizedThreadStart**.
2. Parametrická cieľová metóda.
3. Parametrická verzia preťaženej metódy **Start** pracovného vlákna.

Slabou stránkou použitia parametrického delegáta **ParameterizedThreadStart** a spriaznenej parametrickej metódy je genericita formálneho parametra. Dátovým typom tohto formálneho parametra je **object**, čo znamená, že tento formálny parameter môže byť inicializovaný akoukoľvek hodnotou hodnotového alebo odkazového dátového typu. Ak budeme chcieť striktne zachovať typovú bezpečnosť, môžeme uplatniť iný postup:

1. Deklarujeme novú triedu (s pracovným identifikátorom **X**), ktorá bude obsahovať tieto entity:

    - Dátové členy na uchovanie požadovaných dát, s ktorými budeme chcieť pracovať.

    - Parametrický inštančný konštruktor. Množina formálnych parametrov bude navrhnutá tak, aby dokázala prijať vstupné dáta.

    - Cieľovú bezparametrickú metódu (s pracovným identifikátorom **M**), ktorá bude neskôr spracúvaná na pracovnom vlákne. Napriek tomu, že táto metóda bude bezparametrická, bude môcť pracovať s dátami, pretože tie budú uchované v dátových členoch, ku ktorým bude mať metóda prístup.

2. Založíme inštanciu deklarovanej triedy **X** a náležite ju inicializujeme požadovanými dátami. Dáta poskytneme parametrickému inštančnému konštruktoru triedy.

3. Zhotovíme inštanciu triedy **Thread**. Konštruktoru tejto triedy odovzdáme odkaz na inštanciu bezparametrického delegáta **ThreadStart**, ktorá bude zapuzdrovať odkaz na inštančnú cieľovú metódu (**M**).

Ak opísaný postup aplikujeme na príklade s výpočtom faktoriálu, dopracujeme sa k takémuto obrazu zdrojového kódu:

```
// Deklarácia triedy, ktorá združuje dátový člen a cieľovú metódu.
class Faktoriál
{
    private int n;
    public Faktoriál(int n)
    {
        this.n = n;
    }
    public void VypočítaťFaktoriál()
    {
        Console.WriteLine("Prebieha výpočet faktoriálu...");
        Stopwatch sw = new Stopwatch();
        sw.Start();
        long faktoriál = 1;
        for (int i = 1; i <= n; i++)
        {
            faktoriál *= i;
        }
        sw.Stop();
        Console.WriteLine("Úloha je splnená.");
        Console.WriteLine("{0}! je {1}.", n, faktoriál);
        Console.WriteLine("Exekučný čas: {0} ms.",
            sw.Elapsed.TotalMilliseconds);
    }
}

static void Main(string[] args)
{
    Console.Write("Faktoriál ktorého čísla chcete vypočítať? ");
    int n = Convert.ToInt32(Console.ReadLine());
    Faktoriál faktoriál = new Faktoriál(n);
    Thread pracovnéVlákno = new Thread(
```

```
            new ThreadStart(faktoriál.VypočítaťFaktoriál));
        pracovnéVlákno.Name = "Pracovné vlákno 1";
        pracovnéVlákno.Start();
        pracovnéVlákno.Join();
        Console.Read();
}
```

Komentár k zdrojovému kódu: Trieda **Faktoriál** definuje jeden dátový člen (**n**), parametrický inštančný konštruktor a jednu inštančnú metódu s identifikátorom **VypočítaťFaktoriál**. Po založení objektu triedy **Faktoriál** špecifikujeme člen, ktorého faktoriál chceme vypočítať. Z inštanciačného príkazu triedy **Thread** je zrejmé, že inštancia bezparametrického delegáta **ThreadStart** získa odkaz na inštančnú bezparametrickú cieľovú metódu **VypočítaťFaktoriál**.

Ak bude naším zámerom získať dáta z metódy, ktorá je spracúvaná na pracovnom vlákne, zapracujeme mechanizmus, ktorého podstatou bude vyvolanie metódy spätného volania v okamihu, keď sa skončí činnosť cieľovej metódy. Metóda spätného volania bude aktivovaná pomocou inštancie delegáta. Najskôr uvádzame komplexný zdrojový kód, a potom k nemu zaujmeme stanovisko.

```
public delegate void DelegátSV(int n, long faktoriál);

class Faktoriál
{
    private int n;
    private DelegátSV delegát;
    public Faktoriál(int n, DelegátSV delegát)
    {
        this.n = n;
        this.delegát = delegát;
    }
    public void VypočítaťFaktoriál()
    {
        Console.WriteLine("Prebieha výpočet faktoriálu...");
        Stopwatch sw = new Stopwatch();
        sw.Start();
        long faktoriál = 1;
        for (int i = 1; i <= n; i++)
        {
            faktoriál *= i;
        }
        sw.Stop();
```

```
        Console.WriteLine("Úloha je splnená.");
        Console.WriteLine("Exekučný čas: {0} ms.",
            sw.Elapsed.TotalMilliseconds);

        if (delegát != null)
            delegát(n, faktoriál);
    }
}
class Program
{
    static void Main(string[] args)
    {
        Thread primárneVlákno = Thread.CurrentThread;
        primárneVlákno.Name = "Primárne vlákno";
        Console.Write("Faktoriál ktorého čísla chcete vypočítať? ");
        int n = Convert.ToInt32(Console.ReadLine());
        Faktoriál faktoriál = new Faktoriál(n, new DelegátSV(MetódaSV));
        Thread pracovnéVlákno = new Thread(
            new ThreadStart(faktoriál.VypočítaťFaktoriál));
        pracovnéVlákno.Name = "Pracovné vlákno 1";
        pracovnéVlákno.Start();
        pracovnéVlákno.Join();
        Console.Read();
    }

   public static void MetódaSV(int n, long faktoriál)
    {
        Console.WriteLine("{0}! je {1}.", n, faktoriál);
    }
}
```

Komentár k zdrojovému kódu: Mimo triedu **Faktoriál** deklarujeme parametrického verejne prístupného delegáta s identifikátorom **DelegátSV** (ide o tzv. delegáta spätného volania, ako indikuje skratka „SV“ v identifikátore delegáta). V tele triedy **Faktoriál** definujeme súkromnú dátovú položku s názvom **delegát**, ktorej typom je deklarovaný delegát **DelegátSV**. Táto dátová položka bude inicializovaná odkazom na inštanciu delegáta **DelegátSV**. V signatúre parametrického inštančného konštruktora triedy **Faktoriál** sa nachádzajú definície dvoch formálnych parametrov. Z nášho pohľadu je významný druhý formálny parameter, ktorý očakáva odkaz na inštanciu delegáta spätného volania. Napokon, v tele metódy **VypočítaťFaktoriál** je aktivovaná pomocou inštancie delegáta **DelegátSV** spriaznená metóda spätného volania.

V hlavnej metóde **Main** je významný príkaz, ktorý zakladá inštanciu triedy **Faktoriál**:

```
Faktoriál faktoriál = new Faktoriál(n, new DelegátSV(MetódaSV));
```

Druhým argumentom, ktorý je odovzdávaný parametrickému inštančnému konštruktoru, je odkaz na inštanciu delegáta spätného volania. Ako si môžeme všimnúť, práve vytvorená inštancia delegáta **DelegátSV** je asociovaná s cieľovou metódou. Táto metóda má identifikátor **MetódaSV** a ide o verejne prístupnú, statickú a parametrickú metódu, ktorá je definovaná v primárnej triede riadenej aplikácie.

Exekúcia programu bude nasledujúca:

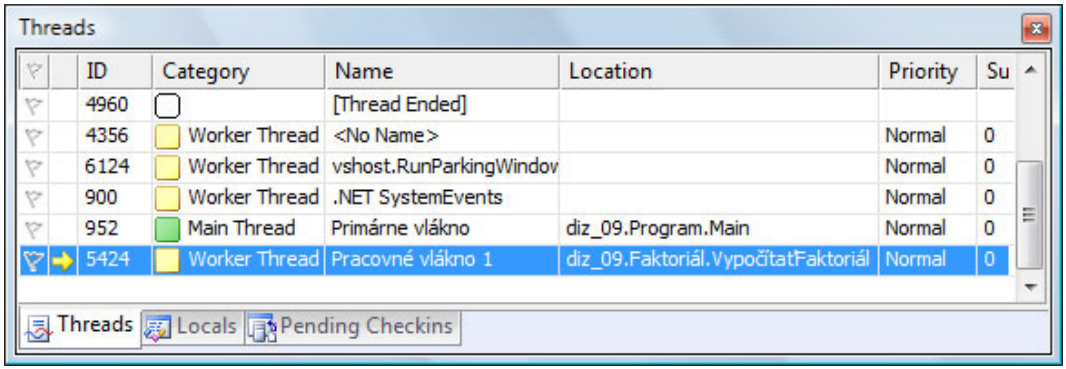
1. Vytvoríme inštanciu triedy **Faktoriál**, ktorú korektne inicializujeme.

2. Dátovú položku **delegát** vytvorenej inštancie inicializujeme odkazom na inštanciu delegáta spätného volania, ktorá obsahuje odkaz na metódu spätného volania.

3. Vytvoríme inštanciu triedy **Thread** a uskutočníme spojenie medzi inštanciou delegáta **ThreadStart** a cieľovou inštančnou metódou **VypočítaťFaktoriál**. Táto metóda bude po spustení pracovného vlákna spracúvaná asynchrónne na tomto vlákne.

4. Keď sa činnosť cieľovej metódy bude chýliť ku koncu, pomocou inštancie delegáta spätného volania aktivujeme požadovanú metódu spätného volania. Metóda spätného volania môže vykonať akúkoľvek činnosť (v našej ukážke zobrazuje na výstupe vypočítanú hodnotu faktoriálu zadaného čísla).

Programové vlákna 2-vláknovej riadenej aplikácie môžeme monitorovať pomocou dialógového okna **Threads** (obr. 12.4).



Obr. 12.4: Programové vlákna 2-vláknovej riadenej aplikácie

13 Fond pracovných programových vlákien

Bázová knižnica vývojovo-exekučnej platformy Microsoft .NET Framework 3.5 deklaruje v mennom priestore **System.Threading** triedu **ThreadPool**. Inštancia tejto triedy pôsobí ako riadený fond pracovných programových vlákien. Fond vlákien reprezentuje súpravu pracovných vlákien, ktoré sú automaticky koordinované virtuálnym správcom fondu pracovných vlákien. To znamená, že vývojári nemusia explicitne vytvárať a spúšťať pracovné vlákna. Dokonca nie je ani nutné, aby programátori priamo riadili životné cykly pracovných vlákien (táto kompetencia náleží správcovi fondu pracovných vlákien).

Vďaka vyššej úrovni abstrakcie, ktorú zavádza fond vlákien, sa vývojári môžu sústrediť na špecifikáciu úloh, ktoré majú byť uskutočnené (tak možno abstrahovať od nutnosti explicitne realizovať technické činnosti spojené s vytváraním a spúšťaním pracovných vlákien).

Správca fondu pracovných vlákien zabezpečí mapovanie požadovaných úloh na optimálny počet pracovných vlákien, ktoré sú získané z fondu pracovných vlákien. Správca fondu pracovných vlákien dokáže pracovné vlákna recyklovať. Spomenutá schopnosť správcu prináša nasledujúce výhody:

1. **Zvýšenie miery opätovnej použiteľnosti pracovných vlákien.** Správca fondu pracovných vlákien recykluje existujúce pracovné vlákna. Konkurenčnou výhodou tohto prístupu je zrušenie relácie typu 1:1 medzi pracovnými vláknami a úlohami, ktoré sú na týchto vláknach uskutočňované. Recyklácia pracovných vlákien zavádza reláciu typu 1:N, v rámci ktorej smie jedno pracovné vlákno počas svojho životného cyklu vykonávať viacero úloh.

2. **Zníženie zaťaženia systému pri vytváraní nových a likvidácii existujúcich pracovných vlákien.** Algoritmy konštrukcie a deštrukcie pracovných vlákien predstavujú najväčšiu záťaž pre virtuálny exekučný systém a operačný systém. Ak bude minimalizovaný výskyt situácií, ktoré si

vynútia spracovanie uvedených časovo a priestorovo intenzívnych operácií, zvýši sa výkonnosť viacvláknovej riadenej aplikácie.

Je dôležité upozorniť na skutočnosť, že všetky pracovné vlákna alokované z fondu pracovných vlákien, sú vláknami v pozadí (ich vlastnosť **IsBackground** je nastavená na logickú pravdu). Ak sa ukončí činnosť aj posledného aktívneho vlákna v popredí, virtuálny exekučný systém v spolupráci so správcom fondu pracovných vlákien zruší všetky pracovné vlákna, ktoré boli pridelené z fondu (vlákna fondu neudržia riadenú aplikáciu nažive).

Každé pracovné vlákno z fondu má implicitne stanovenú alokačnú kapacitu svojho zásobníka a normálnu prioritu. Jeden fyzický proces riadenej aplikácie smie obsahovať práve jeden fond pracovných vlákien. Počet úloh, ktoré budú zasielané fondu vlákien a vzápätí delegované na jednotlivé pracovné vlákna je obmedzený iba veľkosťou inštalovanej fyzickej operačnej pamäte. Fond pracovných vlákien pracuje implicitne s najviac 250 pracovnými vláknami na jeden procesor, resp. na jedno exekučné jadro viacjadrového procesora. Maximálny počet pracovných vlákien môžeme upraviť použitím statickej metódy **SetMaxThreads** triedy **ThreadPool**. Minimálny počet pracovných vlákien bude vždy väčší alebo nanajvýš rovný počtu procesorov počítača, resp. počtu exekučných jadier viacjadrového procesora.

Praktické použitie fondu pracovných vlákien predvedieme na riadenej aplikácii, ktorá nám dovolí súbežne preberať súbory elektronických kníh zo siete Internet. Najskôr uvádzame zdrojový kód programu, a potom k nemu pripojíme komentár.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Net;
using System.Text;
using System.Threading;

namespace diz
{
    // Deklarácia triedy manažéra, ktorý bude uchovávať informácie
```

```
// o zdrojovej a cieľovej lokácii súboru elektronickej knihy.
class Manažér
{
    private string zdrojovýSúbor, cieľovýSúbor;
    public Manažér(string zdrojovýSúbor, string cieľovýSúbor)
    {
        this.zdrojovýSúbor = zdrojovýSúbor;
        this.cieľovýSúbor = cieľovýSúbor;
    }
    public string ZdrojovýSúbor
    {
        get { return zdrojovýSúbor; }
        set { zdrojovýSúbor = value; }
    }
    public string CieľovýSúbor
    {
        get { return cieľovýSúbor; }
        set { cieľovýSúbor = value; }
    }
}

// Deklarácia primárnej triedy programu.
class Program
{
   // Inštanciácia automatickej udalosti ako synchronizačného
   // objektu.
   static AutoResetEvent udalosť = new AutoResetEvent(false);
   // Definícia statickej dátovej položky na uchovanie počtu úloh,
   // ktoré budú realizované pomocou pracovných vlákien
   // získaných z fondu vlákien.
   static int početÚloh;

   static void Main(string[] args)
    {
       // Vytvorenie manažérov úloh. Keďže plánujeme realizovať
       // 3 úlohy, vytvárame 3 manažérov.
        Manažér webovýManažér1 =
           new Manažér("http://www.cl.cam.ac.uk/teaching/" +
            "2000/FoundsCS/Founds-FP.pdf",
            "c:\\Knihy\\Founds-FP.pdf");

        Manažér webovýManažér2 =
           new Manažér("http://home.bway.net/kbeen/teaching/" +
            "intro/resources/introbook.pdf",
            "c:\\Knihy\\Introbook.pdf");

        Manažér webovýManažér3 =
           new Manažér("http://www.haskell.org/tutorial/" +
            "haskell-98-tutorial.pdf", "c:\\Knihy\\Haskell-98.pdf");
```

```
            // Zaslanie úloh správcovi fondu pracovných vlákien.
            ThreadPool.QueueUserWorkItem(
                new WaitCallback(PrevziaťSúborZWebu), webovýManažér1);
            ThreadPool.QueueUserWorkItem(
                new WaitCallback(PrevziaťSúborZWebu), webovýManažér2);
            ThreadPool.QueueUserWorkItem(
                new WaitCallback(PrevziaťSúborZWebu), webovýManažér3);
            udalosť.WaitOne();
            Console.WriteLine("\n************** Všetky úlohy sú hotové. "
                + "**************");
            Console.Read();
        }

        // Definícia metódy, ktorá bude preberať súbory elektronických
        // kníh z Internetu.
        private static void PrevziaťSúborZWebu(object objekt)
        {
            Manažér manažér = (Manažér)objekt;
            Stopwatch sw = new Stopwatch();
            WebClient webovýKlient = new WebClient();
            Console.WriteLine("\n************** Vlákno {0} zahájilo " +
                "operáciu **************",
                Thread.CurrentThread.ManagedThreadId);
            Console.WriteLine("\nPrebieha prevzatie súboru {0}.",
                manažér.ZdrojovýSúbor);
            sw.Start();
            // Synchrónne prevzatie súboru elektronickej knihy.
            webovýKlient.DownloadFile(manažér.ZdrojovýSúbor,
                manažér.CieľovýSúbor);
            sw.Stop();
            FileInfo atribútySúboru = new FileInfo(manažér.CieľovýSúbor);
            Console.WriteLine("\n************** Vlákno {0} ukončilo " +
                "operáciu **************\n",
                Thread.CurrentThread.ManagedThreadId);
            Console.WriteLine("Súbor {0} bol stiahnutý a uložený." +
                "\nČas: " + sw.Elapsed.Seconds + " s" +
                "\nVeľkosť súboru: " + atribútySúboru.Length / 1024 +
                " KB", manažér.CieľovýSúbor);
            // Vláknovo bezpečná inkrementácia hodnoty statickej
            // dátovej položky.
            Interlocked.Increment(ref početÚloh);
            // Ak sú hotové všetky úlohy, meníme stav synchronizačného
            // objektu.
            if(početÚloh == 3)
                udalosť.Set();
        }
    }
}
```

Komentár k zdrojovému kódu: Deklarovaná trieda **Manažér** je pomocnou triedou, ktorej inštancie budú uchovávať informácie o umiestnení zdrojových a cieľových súborov elektronických kníh. V našej ukážke budeme z webu súbežne preberať 3 elektronické knihy, a preto v tele hlavnej metódy **Main** zakladáme a korektne inicializujeme 3 inštancie triedy **Manažér**. Zaslanie požiadavky na prevzatie jedného súboru elektronickej knihy uskutočňujeme volaním statickej metódy **QueueUserWorkItem** triedy **ThreadPool** z menného priestoru **System.Threading**. Táto statická metóda je preťažená: my upotrebujeme jej parametrickú definíciu, ktorej signatúra je naplnená dvomi formálnymi parametrami. Prvý z nich očakáva odkaz na inštanciu delegáta **WaitCallback**. Inštancia delegáta **WaitCallback** bude asociovaná s cieľovou parametrickou metódou, ktorej programový kód bude injektovaný do pracovného vlákna získaného z fondu pracovných vlákien. Druhý formálny parameter statickej metódy **QueueUserWorkItem** je schopný prijať odkaz na akýkoľvek objekt (či už hodnotového, alebo odkazového dátového typu). V prípade potreby môžeme metóde odovzdať odkaz na objekt s dátami.

Z príkazu, ktorý aktivuje charakterizovanú preťaženú verziu statickej metódy **QueueUserWorkItem** vyplýva, že cieľovou metódou je (rovnako statická) metóda **PrevziaťSúborZWebu**. Cieľová metóda bude môcť pracovať s dátovým objektom, ktorý statickej metóde **QueueUserWorkItem** ponúkame ako druhý argument.

Keďže formálny parameter cieľovej metódy **PrevziaťSúborZWebu** je primitívneho odkazového typu **object**, musíme odkaz na dátový objekt podrobiť explicitnej typovej konverzii. V momente, keď získame od používateľa informácie o tom, aký súbor si praje prevziať a kam ho máme umiestniť, voláme inštančnú parametrickú metódu **DownloadFile** inštancie triedy **WebClient** z menného priestoru **System.Net**. Použitie metódy **DownloadFile** je veľmi jednoduché a intuitívne: stačí, ak určíme pozíciu zdrojového súboru (v podobe 1. argumentu) a stanovíme jeho lokalizáciu po transporte (v podobe 2. argumentu).

Aby sme mohli sledovať použitie pracovných vlákien z fondu vlákien, zobrazujeme identifikačné číslo pracovného vlákna. Toto číslo jednoznačne identifikuje pracovné

vlákno získané z fondu vlákien riadenej aplikácie. Rovnako meriame čas, ktorý je potrebný na prevzatie súboru elektronickej knihy. Následne, po uložení už prevzatého súboru, vypisujeme informačnú správu o jeho alokačnej kapacite.

Vzhľadom na to, že chceme používateľa informovať o dokončení spracovania všetkých úloh, pracujeme s automatickou udalosťou, resp. s inštanciou triedy **AutoResetEvent**, ako so synchronizačno-signalizačným objektom. Tento objekt nám umožní signalizovať stav, keď budú dokončené všetky 3 úlohy, ktoré pomocou prostriedkov fondu pracovných vlákien uskutočňujeme. Len čo sa tak stane, voláme metódu **Set** automatickej udalosti, čím jej prikážeme vyslať signál. Na tento signál čaká hlavná metóda **Main**, pretože v jej tele voláme metódu **WaitOne** objektu automatickej udalosti.

V tele cieľovej metódy **PrevziaťSúborZWebu** vykonávame vláknovo bezpečnú inkrementáciu statickej dátovej položky pomocou statickej metódy **Increment** triedy **Interlocked** z menného priestoru **System.Threading**.

Po spustení riadenej aplikácie zaregistrujeme uskutočnenie týchto akcií:

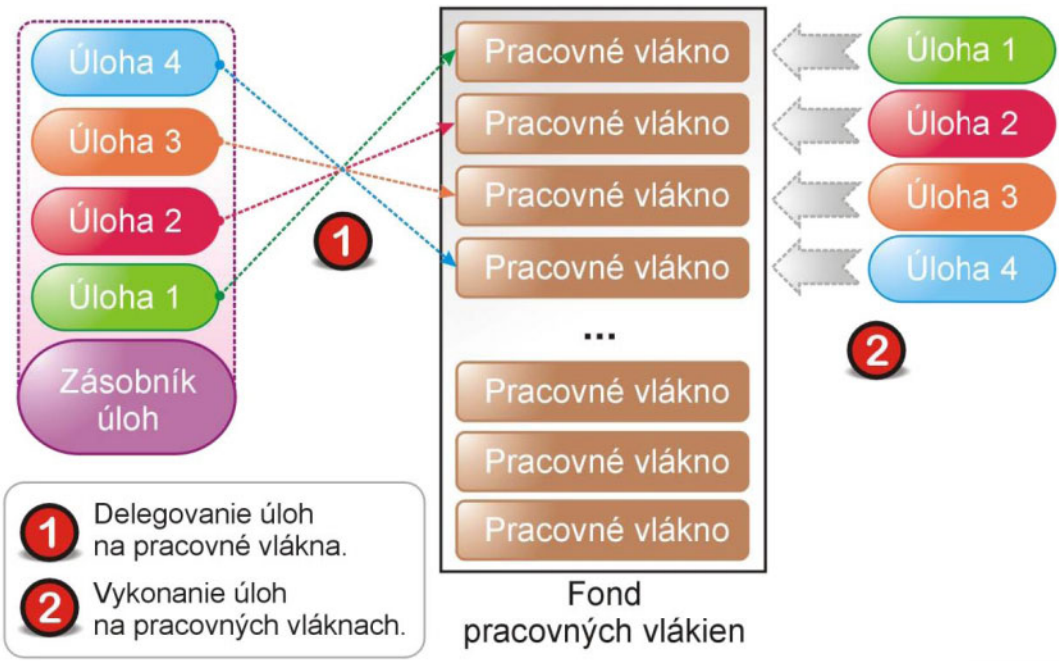
1. Správca fondu pracovných vlákien prijme žiadosti o vykonanie 3 úloh (prevzatie 3 elektronických kníh).

2. Správca fondu pracovných vlákien vyberie z množstva disponibilných pracovných vlákien 3 vlákna a do každého injektuje kód cieľovej metódy.

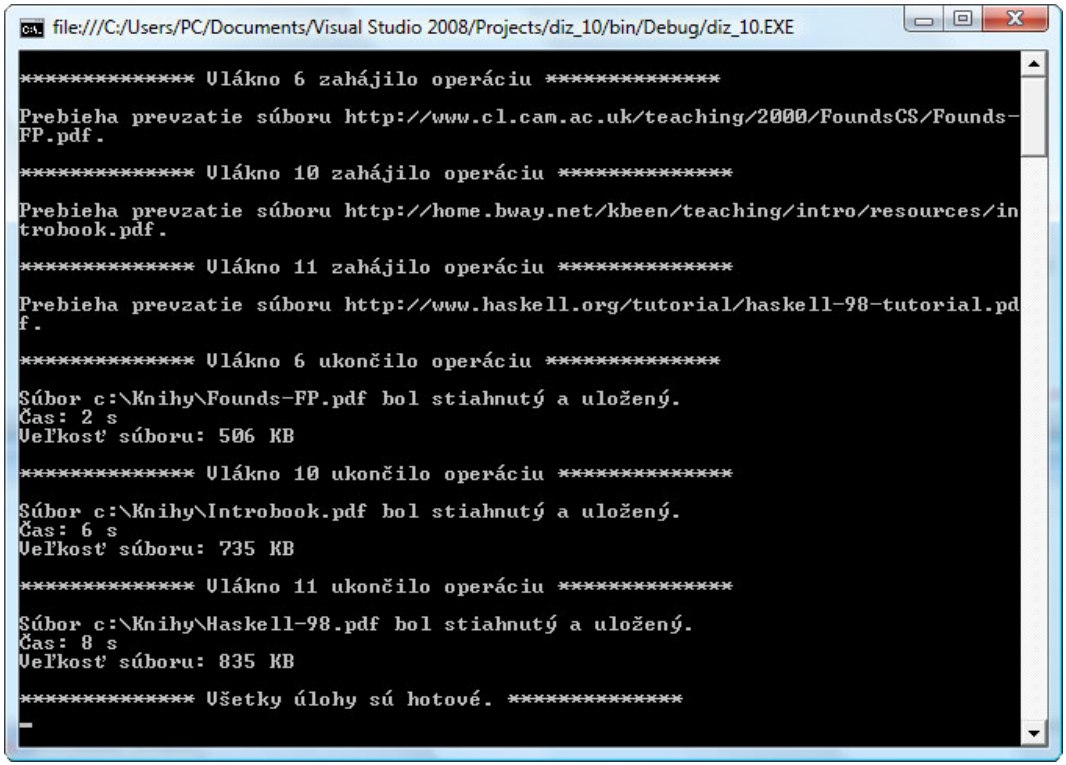
3. Pracovné vlákna začnú plniť delegované úlohy.

Na obr. 13.1 je znázornená kooperácia medzi programovými úlohami a pracovnými vláknami fondu pracovných vlákien.

Výstup riadenej aplikácie, využívajúcej pracovné vlákna z fondu vlákien, je zobrazený na obr. 13.2.



Obr. 13.1: Asociácia medzi úlohami a pracovnými vláknami získanými z fondu pracovných vlákien



Obr. 13.2: Výstup riadenej aplikácie, ktorá pri svojej práci využíva fond pracovných vlákien

14 Synchronizácia programových vlákien a synchronizačné primitíva

Pri programovaní viacvláknových aplikácií musíme dbať na synchronizáciu pracovných činností, ktoré programové vlákna uskutočňujú vtedy, keď pracujú so zdieľanými prostriedkami. Ak by sme totiž nesynchronizovali pracovné modely programových vlákien, ich paralelná exekúcia by mohla spôsobiť konkurenčné prístupy k zdieľaným prostriedkom s rizikom zanechania týchto prostriedkov v nekonzistentnom stave. Pre účely tejto publikácie budeme skúmať synchronizáciu programových vlákien, ktoré pracujú so zdieľanými prostriedkami. Týmito prostriedkami sú najmä statické inštancie hodnotových a odkazových dátových typov, a to tak primitívnych, ako aj používateľsky deklarovaných.

Viacvláknová riadená aplikácia obsahuje n vlákien, ktoré okupujú adresový priestor aplikačnej domény fyzického procesu[22]. Keďže jednotlivé programové vlákna zdieľajú adresový priestor aplikačnej domény, môžu pristupovať k akýmkoľvek prostriedkom, ktoré sa v tomto priestore nachádzajú. Pre úplnosť dodajme, že každé programové vlákno obsahuje vlastný zásobník, do ktorého sú ukladané automatické objekty hodnotových dátových typov. Spomenuté automatické objekty nie sú predmetom synchronizácie, pretože sú lokálne pre každé programové vlákno (a nie sú teda zdieľané viacerými vláknami súčasne).

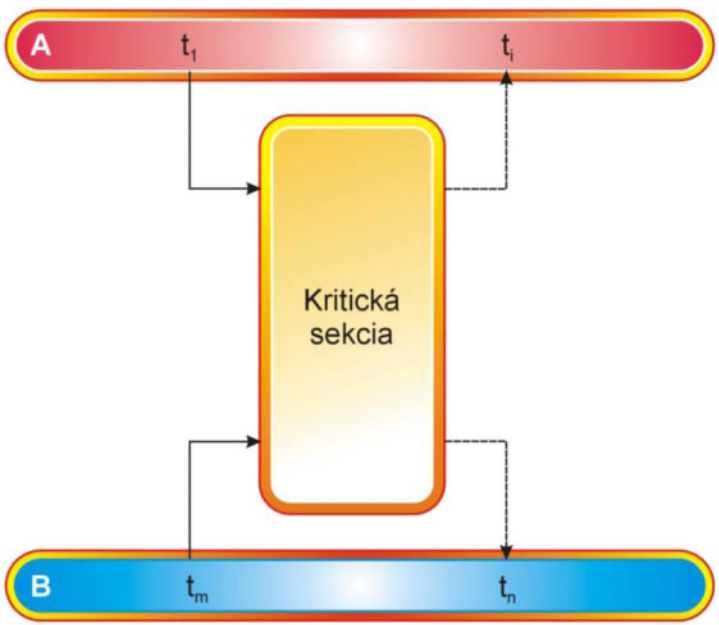
Synchronizácia prístupu viacerých programových vlákien k zdieľaným prostriedkom sa uskutočňuje pomocou synchronizačných primitív. Synchronizačné primitíva sú špeciálne synchronizačné objekty, ktoré garantujú exkluzívny, tzv. vzájomne výlučujúci sa prístup k zdieľanému prostriedku počas istého časového intervalu. Časový interval, počas ktorého trvá exkluzivita prístupu práve jedného programového vlákna k zdieľanému prostriedku, je spravidla determinovaný kritickou sekciou. Kritická sekcia predstavuje blok programového kódu, ktorý môže

[22] V našich ďalších úvahách budeme predpokladať, že všetky programové vlákna riadenej aplikácie sú situované v jednej aplikačnej doméne fyzického procesu tejto aplikácie. Abstrahujeme teda od situácií, kedy by programové vlákna boli rozmiestnené naprieč viacerými aplikačnými doménami fyzického procesu riadenej aplikácie.

v danom okamihu vykonávať práve jedno programové vlákno. Ak sa jedno programové vlákno aktuálne nachádza v kritickej sekcii, ďalšie vlákno nemôže do kritickej sekcie vstúpiť skôr, než ju opustí prvé vlákno. Z uvedeného vyplýva, že synchronizácia eliminuje pravdepodobnosť súbežného prístupu viacerých programových vlákien k zdieľanému prostriedku v kritickej sekcii.

Okrem toho však synchronizácia súvisí aj s tranzitom medzi paralelnou a sekvenčnou exekúciou programového bloku, ktorý tvorí kritickú sekciu. Hoci pred vstupom do kritickej sekcie mohla byť aplikácia spracúvaná paralelne, v kritickej sekcii sa mení jej exekučný model na rýdzo sekvenčný (vykonanie kódu kritickej sekcie môže vo vymedzenom časovom intervale realizovať práve jedno programové vlákno).

Z hľadiska efektívnej exekúcie programových inštrukcií viacvláknovej riadenej aplikácie je žiaduce, aby bol čas alokovaný na spracovanie kritickej sekcie čo možno najkratší.



Obr. 14.1: Kritická sekcia

Komentár k obr. 14.1: Obrázok ukazuje synchronizovaný prístup dvoch programových vlákien (s identifikátormi **A** a **B**) k programovému bloku, ktorý je determinovaný kritickou sekciou. Vlákno **A** vstupuje do kritickej sekcie v čase t_1. V tomto čase začína vlákno **A** vykonávať príkazy kritickej sekcie. Exekúcia príkazov kritickej sekcie vláknom **A** sa končí v čase t_i, kedy vlákno **A** opúšťa kritickú sekciu. Podotknime, že v čase danom intervalom $<t_1, t_i>$ nemôže do kritickej sekcie vstúpiť žiadne iné vlákno, pretože exkluzívnym prístupom k nej disponuje vlákno **A**. V čase t_m (pričom platí, že $t_m > t_i$) vchádza do kritickej sekcie vlákno **B**. Toto vlákno spracúva príkazy kritickej sekcie počas trvania časového intervalu $<t_m, t_n>$. V čase t_n vlákno **B** opúšťa kritickú sekciu. Analogicky, pokiaľ neuplynie časový interval $<t_m, t_n>$, žiadne iné programové vlákno nesmie vstúpiť do kritickej sekcie. Dôvod je evidentný: kým nevyprší spomenutý časový interval, exkluzívny prístup do kritickej sekcie má iba vlákno **B**.

Synchronizácia prístupu k zdieľaným prostriedkom je prospešná, pretože nám umožňuje vyhnúť sa vzniku veľmi nebezpečných chýb, ku ktorým patria najmä preteky vlákien a uviaznutia[23].

14.1 Preteky vlákien

Pri pretekoch vlákien je výstup paralelného programu nedeterministický, pretože závisí od poradia operácií vykonávaných viacerými programovými vláknami nad nesynchronizovanými zdieľanými prostriedkami. V závislosti od toho, ktoré programové vlákno a v akom čase vykoná operáciu so zdieľanými prostriedkom, smie program produkovať rôzne výstupy, a to pri identickej súprave vstupných dát.

Problémy, ktoré sa objavujú pri pretekoch vlákien, budeme demonštrovať na nasledujúcom príklade.

[23] Preteky vlákien, resp. uviaznutia sú v origináli nazývané termínmi „race conditions“, resp. „deadlock“.

Príklad: Máme dve programové vlákna **A** a **B**, ktoré budú manipulovať s dátovým členom **x** statického objektu **o**. Dátový člen **x** statického objektu je explicitne inicializovaný diskrétnou celočíselnou konštantou 60 ešte predtým, ako dôjde k spusteniu programových vlákien **A** a **B**. Vlákno **A** modifikuje dátový člen **x** tak, že k jeho aktuálnej hodnote pripočíta hodnotu 10, a vzápätí hodnotu súčtu uloží späť do dátového člena (vlákno **A** teda spracúva príkaz o.x+=10;). Vlákno **B** modifikuje dátový člen **x** tak, že jeho aktuálnu hodnotu delí dvomi a podiel vzápätí priradí do dátového člena (vlákno **B** teda vykonáva operáciu o.x/=2;).

Ak ako prvé spustíme vlákno **A** a ako druhé spustíme vlákno **B**, budeme zrejme chcieť, aby najskôr prebehla modifikácia dátového člena statického objektu vláknom **A**, a potom sa uskutočnila modifikácia dátového člena statického objektu vláknom **B**. Túto situáciu zachytáva tab. 14.1.

Poradie operácií	**Vlákno A** (o.x+=10;)	**Vlákno B** (o.x/=2;)	**Dátový člen statického objektu (o.x)**
1.	Načítať(o.x): 60		60
2.	Modifikovať(o.x): 70		60
3.	Uložiť(o.x): 70		70
4.		Načítať(o.x): 70	70
5.		Modifikovať(o.x): 35	70
6.		Uložiť(o.x): 35	**35**

Tab. 14.1: Preteky vlákien (situácia č. 1 – želateľný stav, implicitná synchronizácia)

V okamihu, keď sú obe programové vlákna hotové s úpravou stavu statického objektu, má jeho dátový člen hodnotu 35. Analyzovaná situácia reprezentuje želateľný stav, pretože modifikácie dátového člena statického objektu realizované programovými vláknami sa neprekrývajú, ale sú uskutočňované v správnom (môžeme povedať implicitne synchronizovanom) poradí.

Vzhľadom na to, že prístupy programových vlákien **A** a **B** k dátovému členu statického objektu nie sú zatiaľ explicitne synchronizované, môžu sa vyskytnúť ďalšie situácie, v ktorých bude finálna hodnota dátového člena **x** závisieť od poradia,

v akom budú vlákna s týmto členom manipulovať. Za predpokladu, že paralelná exekúcia bude zahájená vláknom **A**, dospejeme k výsledkom, ktoré sú uvedené v tab. 14.2.

Poradie operácií	**Vlákno A** (o.x+=10;)	**Vlákno B** (o.x/=2;)	**Dátový člen statického objektu (o.x)**
1.	Načítať(o.x): 60		60
2.	Modifikovať(o.x): 70	Načítať(o.x): 60	60
3.	Uložiť(o.x): 70	Modifikovať(o.x): 30	70
4.		Uložiť(o.x): 30	**30**

Tab. 14.2: Preteky vlákien (situácia č. 2 – nebezpečný stav)

Prístup k statickému objektu nie je explicitne synchronizovaný, a preto sa vyskytujú preteky programových vlákien, ktoré zapríčiňujú nekonzistentný finálny stav tohto objektu. Vlákno **A** v prvej operácii načíta inicializačnú hodnotu dátového člena statického objektu, no kým ju stačí modifikovať a aktualizovať, vlákno **B** načíta pôvodnú hodnotu dátového člena statického objektu, ktorý je aktuálne modifikovaný vláknom **A**. Ako vidíme, vlákno **B** nachádza dátový člen statického objektu uprostred modifikačného procesu, ktorý je na tento člen aplikovaný vláknom **A**. Povedané inak, vlákno **B** zachytáva dátový člen statického objektu v dátovo nesúdržnom stave. Pri ďalšej analýze dospejeme k tomuto záveru: Hodnota (70), ktorú vlákno **A** uloží do dátového člena statického objektu v 3. kroku, bude prepísaná hodnotou (30), ktorú uloží do tohto člena vlákno **B** v 4. kroku.

Ak zmeníme poradie spustenia vlákien, zistíme, že preteky vlákien môžu generovať ďalšiu problematickú situáciu (tab. 14.3).

Poradie operácií	Vlákno A (o.x+=10;)	Vlákno B (o.x/=2;)	Dátový člen statického objektu (o.x)
1.		Načítať(o.x): 60	60
2.	Načítať(o.x): 60	Modifikovať(o.x): 30	60
3.	Modifikovať(o.x): 70	Uložiť(o.x): 30	30
4.	Uložiť(o.x): 70		**70**

Tab. 14.3: Preteky vlákien (situácia č. 3 – nebezpečný stav)

Za týchto okolností diagnostikujeme, že v dátovom člene statického objektu je po spracovaní modifikácií uložená hodnota 70. Opakuje sa podobný scenár, ako v predchádzajúcom prípade, len s tým rozdielom, že v tomto kontexte to je programové vlákno **A**, ktoré pracuje s nekonzistentnou hodnotou dátového člena statického objektu.

Správanie paralelného programu, v ktorom sa objavujú preteky vlákien, je nepredvídateľné, pretože hodnota konkrétneho výstupu je ovplyvnená viacerými skutočnosťami, napríklad internými časovacími mechanizmami plánovača úloh operačného systému, modelom paralelnej exekúcie či prioritami programových vlákien pri pseudoparalelnej exekúcii.

Problémy, ktoré prinášajú preteky vlákien, sa pomerne ťažko identifikujú, pretože v mnohých reláciách môže zdanlivo korektný paralelný program produkovať správne výsledky. Zastúpenie chybných výstupov je variabilné a navyše sa vyznačuje minimálnou mierou možnej predikcie, takže softvéroví vývojári nie sú schopní vopred určiť relatívnu početnosť výskytu chybových stavov. Tento ukazovateľ vieme zistiť iba pomocou empirickej analýzy.

Riešenie problému pretekov programových vlákien spočíva v synchronizovanom prístupe jednotlivých vlákien k zdieľanému prostriedku (v našom prípade k dátovému členu statického objektu). Za týmto účelom použijeme jedno zo synchronizačných primitív a nariadime synchronizovaný prístup k dátovému členu statického objektu. Z množiny synchronizačných primitív si vyberieme monitor, čo je synchronizačný objekt, ktorý zabezpečí požadovanému vláknu exkluzívny prístup

k statickému objektu počas determinovaného časového intervalu exekúcie kritickej sekcie. Celá operácia bude prebiehať takto:

1. Vlákno **A** použije monitor na získanie zámku k statickému objektu. Zámok garantuje výhradný prístup vlákna **A** k statickému objektu počas trvania kritickej sekcie. Kritická sekcia je blok kódu, v ktorom bude vlákno **A** uskutočňovať modifikáciu hodnoty dátového člena statického objektu. Kým vlákno **A** drží pomocou monitora zámok na statickom objekte, nie je možné, aby iné vlákno súbežne pristupovalo k tomuto objektu. Ostatné vlákna musia čakať, dokiaľ vlákno **A** neuvoľní zámok nad statickým objektom.

2. Vlákno **A** upraví hodnotu dátového člena statického objektu podľa požadovaného vzoru, pričom triáda operácií načítanie/modifikácia/uloženie bude realizovaná bez rizika prerušenia iným programovým vláknom.

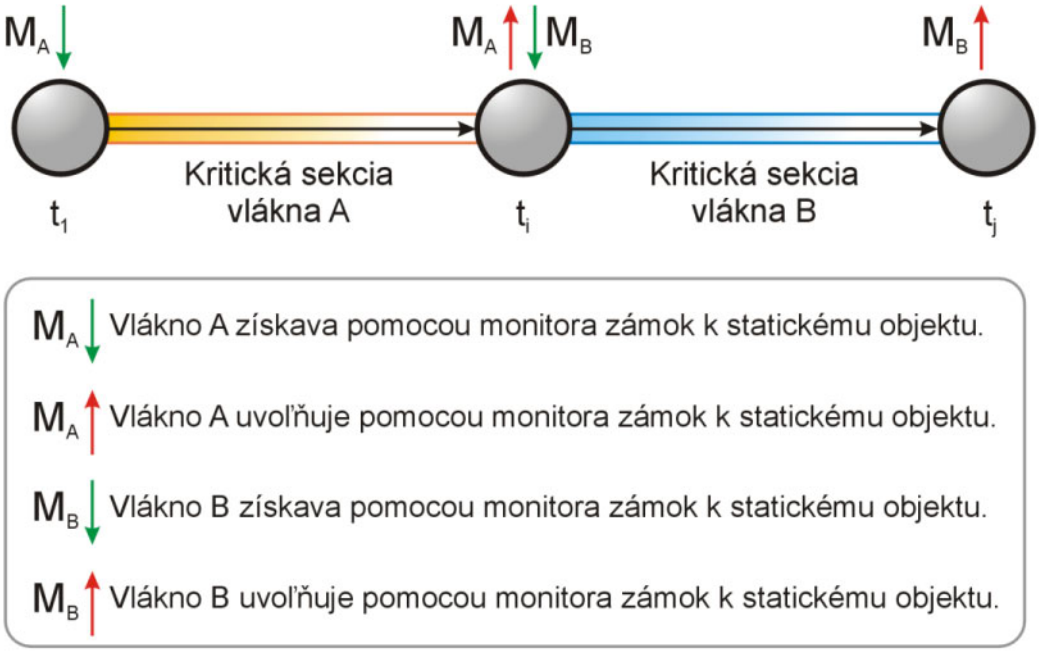
3. Po modifikácii stavu statického objektu vlákno **A** pomocou monitora uvoľní zámok k tomuto objektu. Po uvoľnení zámku a opustení kritickej sekcie vláknom **A** smie k statickému objektu získať prístup (opäť pomocou monitora) vlákno **B** a vykonať s týmto objektom zamýšľané operácie vo svojej kritickej sekcii.

Schematický model synchronizovaného prístupu k dátovému členu statického objektu programovými vláknami **A** a **B** ukazuje tab. 14.4.

Poradie operácií	**Vlákno A** (o.x+=10;)	**Vlákno B** (o.x/=2;)	**Dátový člen statického objektu (o.x)**
1.	Získať zámok pomocou monitora.	Čakanie na získanie zámku.	60
2.	Načítať(o.x): 60		60
3.	Modifikovať(o.x): 70		60
4.	Uložiť(o.x): 70		70
5.	Uvoľniť zámok pomocou monitora.		70
6.		Získať zámok pomocou monitora.	70
7.		Načítať(o.x): 70	70
8.		Modifikovať(o.x): 35	70
9.		Uložiť(o.x): 35	35
10.		Uvoľniť zámok pomocou monitora.	**35**

Tab. 14.4: Preteky vlákien (situácia č. 1 – želateľný stav, explicitná synchronizácia pomocou monitora)

Na obr. 14.2 ukazujeme, ako vyzerá synchronizovaný prístup dvoch programových vlákien k statickému objektu prostredníctvom monitorov.



Obr. 14.2: Synchronizovaný prístup viacerých vlákien
k statickému objektu pomocou monitorov

14.2 Praktický príklad detekcie a korekcie pretekov vlákien v jazyku C# 3.0

Program, ktorý sme vytvorili v jazyku C# 3.0, vytvára dve pracovné programové vlákna. Obe pracovné vlákna operujú so statickým jednorozmerným poľom celých čísel. 1. pracovné vlákno vykonáva inkrementáciu hodnôt všetkých prvkov statického poľa, zatiaľ čo 2. pracovné vlákno sa koncentruje na dekrementáciu hodnôt všetkých prvkov statického poľa. Statické pole je v rámci svojej definície inicializované aritmetickou postupnosťou celých čísel z intervalu <1, 5> s diferenciou 1. Keďže inkrementácia a dekrementácia sú vzájomne opačné aritmetické operácie, očakávame, že po skončení činnosti oboch pracovných vlákien

bude dátový obsah statického poľa zhodný s jeho pôvodným zložením, ktoré sme poľu prisúdili v čase definičnej inicializácie.

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

namespace diz
{
    class Program
    {
        // Definičná inicializácia statického jednorozmerného poľa.
        static int[] pole = {1, 2, 3, 4, 5};

        static void Main(string[] args)
        {
            // Vytvorenie 2-prvkového poľa pracovných vlákien.
            Thread[] poleVlákien = {
                new Thread(new ThreadStart(Inkrementovať)),
                new Thread(new ThreadStart(Dekrementovať))
            };
            Console.WriteLine("Operácia bola zahájená.");

            // Spustenie pracovných vlákien.
            poleVlákien[0].Start();
            poleVlákien[1].Start();

            // Čakanie na dokončenie činností pracovných vlákien.
            poleVlákien[0].Join();
            poleVlákien[1].Join();

            Console.WriteLine("Operácia je hotová.");

            // Kontrola korektnosti modifikácie poľa.
            int súčet = 0;
            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine("{0}. prvok poľa: {1}.", i + 1,
                    pole[i]);
                súčet += pole[i];
            }
            if (súčet != 15)
                Console.WriteLine("Chybná modifikácia poľa.");
            else
                Console.WriteLine("Správna modifikácia poľa.");
```

```
            Console.Read();
        }

        // Definícia metódy 1. pracovného vlákna.
        static void Inkrementovať()
        {
            for (int i = 0; i < 5; i++)
            {
               pole[i]++;
               Console.WriteLine("[PV1-Inkrementácia]: " +
                    "{0}. prvok poľa: {1}.", i + 1, pole[i]);
                Thread.Sleep(500);
            }
        }

        // Definícia metódy 2. pracovného vlákna.
        static void Dekrementovať()
        {

            for (int i = 0; i < 5; i++)
            {
                pole[i]--;
                Console.WriteLine("[PV2-Dekrementácia]: " +
                    "{0}. prvok poľa: {1}.", i + 1, pole[i]);
                Thread.Sleep(500);
            }
        }
    }
}
```

Komentár k zdrojovému kódu: Princíp fungovania 2-vláknového programu je zrejmý. Len čo vytvoríme statické pole, zostrojujeme dvojicu pracovných vlákien, ktoré uskutočňujú vzájomné rušiace sa aritmetické modifikácie hodnôt prvkov poľa. Keď ktorékoľvek z pracovných vlákien vykoná predpísanú transformáciu, odošle do výstupného dátového prúdu správu o aktuálnej hodnote editovaného prvku statického poľa. Medzi modifikáciami jednotlivých prvkov statického poľa vlákna čakajú 0,5 sekundy (na tento časový interval uvádzame vlákna do režimu spánku).

Vo chvíli, keď sú pracovné vlákna hotové s inkrementačno-dekrementačnými operáciami, zahajuje primárne vlákno kontrolu správnosti modifikácie statického poľa. Ako sme už uviedli, po spracovaní modifikácií by mal byť obsah poľa taký istý ako v čase jeho vytvorenia. Súčet prvkov statického poľa by sa teda mal rovnať

hodnote 15. Ak primárne vlákno zistí pri kontrole inú súčtovú hodnotu, zobrazuje správu o chybnej modifikácii statického poľa.

Analyzovaný program je chybný, pretože neimplementuje žiaden mechanizmus synchronizácie prístupu pracovných vlákien k statickému poľu. Problém spočíva v tom, že pracovné vlákna súperia o to, ktoré z nich ako prvé pristúpi k určitému prvku statického poľa za účelom aplikovania aritmetickej operácie. Vzhľadom na absentujúcu synchronizáciu sa môže stať, že jedno z pracovných vlákien získa prístup k prvku statického poľa, ktoré je však v danej chvíli modifikované iným vláknom. Modifikácia v našom ponímaní predstavuje inkrementáciu alebo dekrementáciu hodnoty prvku statického poľa. Hoci pre softvérového vývojára v jazyku C# 3.0 je inkrementácia, resp. dekrementácia atomickou operáciou[24], v skutočnosti to tak nie je. Tak inkrementácia, ako aj dekrementácia je triádou nasledujúcich operácií:

1. Získanie hodnoty prvku statického poľa z operačnej pamäte a jej uchovanie v registri exekučného jadra viacjadrového procesora.

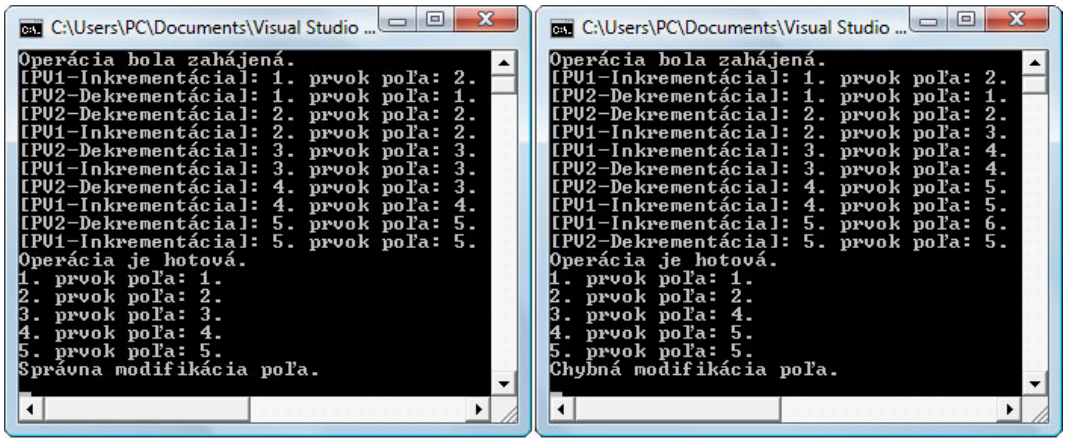
2. Modifikácia hodnoty prvku statického poľa a jej uloženie do registra exekučného jadra viacjadrového procesora.

3. Uloženie modifikovanej hodnoty prvku statického poľa z registra exekučného jadra viacjadrového procesora do operačnej pamäte.

Pri pretekoch vlákien dochádza k závažnej chybe vo výpočtových procesoch vtedy, keď jedno z pracovných vlákien začne pracovať s prvkom statického poľa predtým, ako iné pracovné vlákno dokončí celú triádu operácií s týmto prvkom poľa. Pretože pracovné vlákno získava nekorektnú hodnotu prvku statického poľa, výsledok jeho práce bude rovnako nesprávny. Ak nastane v dôsledku pretekov vlákien chybná modifikácia prvkov statického poľa, konečný súčet hodnôt týchto prvkov nebude

[24] Atomická operácia je nedeliteľná operácia, ktorá je vždy vykonaná ako celok bez rizika prerušenia.

reflektovať správnu hodnotu 15. Obr. 14.3 ukazuje dva výstupy 2-vláknového programu, pričom jeden z nich je správny a druhý chybný.



Obr. 14.3: 2-vláknový program, v ktorom preteky vlákien spôsobujú chybný výstup (vpravo)

Najväčším rizikom pretekov vlákien je nedeterministická povaha programu, ktorá môže v závislosti od načasovania pracovných vlákien vyústiť do správneho, ale aj chybného spracovania programu. Počas empirickej analýzy sme dospeli k výsledkom, ktoré sú zoskupené v tab. 14.5.

Kolekcie spustenia programu							
Relácia	1. kolekcia	Relácia	2. kolekcia	Relácia	3. kolekcia	Relácia	4. kolekcia
1.	✓	1.	✓	1.	✓	1.	✗
2.	✓	2.	✗	2.	✓	2.	✓
3.	✓	3.	✓	3.	✓	3.	✓
4.	✗	4.	✓	4.	✓	4.	✓
5.	✓	5.	✓	5.	✓	5.	✓

Kolekcie spustenia programu							
Relácia	1. kolekcia	Relácia	2. kolekcia	Relácia	3. kolekcia	Relácia	4. kolekcia
6.	✓	6.	✗	6.	✓	6.	✗
7.	✓	7.	✓	7.	✓	7.	✗
8.	✓	8.	✓	8.	✓	8.	✓
9.	✓	9.	✓	9.	✓	9.	✓
10.	✓	10.	✓	10.	✓	10.	✓
R(Fx)	10 %		20 %		0 %		30 %

Tab. 14.5: Empirická analýza detekcie nekorektných výstupov 2-vláknového programu s nedeterminizmom, ktorý spôsobujú preteky vlákien

Uskutočnili sme 4 kolekcie testov paralelného programu s neošetrenou chybou, ktorú spôsobujú preteky vlákien. V každej kolekcii sme sledovali 10 relácií, počas ktorých sme paralelný program spustili a zaznamenali jeho výstup. Ak bol výstup v poriadku, označili sme spustenie programu v danej relácii symbolom ✓. Naopak, ak výstup nebol korektný, relácii programu sme prisúdili symbol ✗. V každej kolekcii testov sme vypočítali relatívnu početnosť výskytu chybových stavov, reprezentovanú ukazovateľom R(Fx). Zo všetkých 4 kolekcií testov, iba jediná (3. v poradí) vykazovala nulovú početnosť vzniku chybových stavov. V ostatných kolekciách testov sa chyby objavovali s rôznou variabilitou, od 10 % do 30 %.

Problémy, ktoré vznikajú ako priamy dôsledok pretekov vlákien, môžeme riešiť pomocou monitora, špeciálneho synchronizačného objektu. Najskôr uvedieme opravený zdrojový kód, a potom k nemu pripojíme komentár. Modifikované partie sú zvýraznené sivou podkladovou farbou.

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

namespace diz
{
    class Program
    {
        // Definičná inicializácia statického jednorozmerného poľa.
        static int[] pole = { 1, 2, 3, 4, 5 };

        static void Main(string[] args)
        {
            // Vytvorenie 2-prvkového poľa pracovných vlákien.
            Thread[] poleVlákien = {
                new Thread(new ThreadStart(Inkrementovať)),
                new Thread(new ThreadStart(Dekrementovať))
            };
            Console.WriteLine("Operácia bola zahájená.");

            // Spustenie pracovných vlákien.
            poleVlákien[0].Start();
            poleVlákien[1].Start();

            // Čakanie na dokončenie činností pracovných vlákien.
            poleVlákien[0].Join();
            poleVlákien[1].Join();

            Console.WriteLine("Operácia je hotová.");

            // Kontrola korektnosti modifikácie poľa.
            int súčet = 0;
            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine("{0}. prvok poľa: {1}.", i + 1,
                    pole[i]);
                súčet += pole[i];
            }
            if (súčet != 15)
                Console.WriteLine("Chybná modifikácia poľa.");
            else
                Console.WriteLine("Správna modifikácia poľa.");
            Console.Read();
        }

        // Definícia metódy 1. pracovného vlákna.
        static void Inkrementovať()
```

```
        {
            for (int i = 0; i < 5; i++)
            {
                // Pomocou monitora umiestnime zámok na statické pole
                // a vstúpime do kritickej sekcie.
                Monitor.Enter(pole);
                try
                {
                    // Vláknovo bezpečná inkrementácia hodnoty prvku
                    // statického poľa.
                    pole[i]++;
                }
                // Pomocou monitora uvoľníme zámok nad statickým poľom
                // a opustíme kritickú sekciu.
                finally
                {
                    Monitor.Exit(pole);
                }
                Console.WriteLine("[PV1-Inkrementácia]: " +
                    "{0}. prvok poľa: {1}.", i + 1, pole[i]);
                Thread.Sleep(500);
            }
        }

        // Definícia metódy 2. pracovného vlákna.
        static void Dekrementovať()
        {
            for (int i = 0; i < 5; i++)
            {
                // Pomocou monitora umiestnime zámok na statické pole
                // a vstúpime do kritickej sekcie.
                Monitor.Enter(pole);
                try
                {
                    // Vláknovo bezpečná dekrementácia hodnoty prvku
                    // statického poľa.
                    pole[i]--;
                }
                // Pomocou monitora uvoľníme zámok nad statickým poľom
                // a opustíme kritickú sekciu.
                finally
                {
                    Monitor.Exit(pole);
                }
                Console.WriteLine("[PV2-Dekrementácia]: " +
                    "{0}. prvok poľa: {1}.", i + 1, pole[i]);
                Thread.Sleep(500);
            }
        }
```

```
        }
}
```

Komentár k zdrojovému kódu: Na syntaktickej úrovni je monitor čoby synchronizačné primitívum reprezentovaný inštanciou triedy **Monitor** z menného priestoru **System.Threading**. Hoci monitor musí existovať, nezískame ho inštanciáciou triedy **Monitor**, ale volaním statickej metódy **Enter** tejto triedy[25]. Metóda **Enter** je parametrická a prijíma odkaz na objekt, na ktorý chceme umiestniť pomocou monitora zámok. Volanie statickej metódy **Enter** ohraničuje blok, ktorý pôsobí ako kritická sekcia. Ak jedno pracovné vlákno umiestni na objekt zámok, iné pracovné vlákno nemôže s týmto objektom pracovať skôr, než prvé pracovné vlákno zámok z objektu neuvoľní. Objekt opatrený zámkom sa nachádza v kritickej sekcii tak dlho, pokiaľ nie je aktivovaná parametrická statická metóda **Monitor.Exit**. Keď túto metódu zavoláme a poskytneme jej odkaz na dosiaľ uzamknutý objekt, monitor zabezpečí odomknutie objektu a zanechanie kritickej sekcie.

V našom paralelnom programe musíme monitormi ošetriť telá cyklov **for** situovaných v metódach pracovných vlákien. Najskôr volaním metódy **Monitor.Enter** získame monitor, ktorý uzamkne statické pole. Pokračujeme vstúpením do štruktúrovaného bloku **try**, v ktorom uskutočňujeme vláknovo bezpečnú aritmetickú operáciu s požadovaným prvkom statického poľa. Napokon spracujeme blok **finally**, v ktorom voláme metódu **Monitor.Exit**, čím uvoľňujeme zámok z objektu a vystupujeme z kritickej sekcie.

Je dôležité, aby sme raz uzamknutý objekt vo vhodnej chvíli správne odomkli. Z tohto dôvodu vkladáme volanie metódy **Monitor.Exit** do bloku **finally**, čím máme zaručené, že k uvoľneniu zámku dôjde aj vtedy, ak by programové príkazy v bloku **try** spôsobili generovanie chybovej výnimky.

Po zapracovaní monitorov máme záruku, že paralelný program bude vždy produkovať správne výstupy, pretože prístup pracovných vlákien k statickému poľu je synchronizovaný.

[25] Triedu **System.Threading.Monitor** nie je možné podrobiť inštanciačnému procesu, pretože je statická.

Z technického hľadiska si dovoľujeme pripojiť nasledujúce konštatovanie: Trieda **Monitor** z menného priestoru **System.Threading** umožňuje implicitne mapovať monitory iba na inštancie odkazových dátových typov. Snaha aplikovať monitor na inštanciu (premennú) hodnotového dátového typu je sprevádzaná aktiváciou mechanizmu zjednotenia typov. Výsledkom práce mechanizmu zjednotenia typov je alokovanie objektovej skrinky na riadenej halde a vloženie hodnoty premennej hodnotového typu do tejto skrinky. Čo nie je na prvý pohľad zrejmé, je fakt, že mechanizmus zjednotenia typov je celkovo spustený dvakrát: prvýkrát pri volaní statickej metódy **Monitor.Enter** a druhýkrát pri volaní statickej metódy **Monitor.Exit**. Tento pracovný model je nebezpečný, pretože v skutočnosti neposkytuje žiadnu synchronizáciu objektu (premennej hodnotového typu), ktorý sme pôvodne chceli synchronizovať.

Pre úplnosť výkladu poznamenajme, že rovnakú funkcionalitu, akú nám ponúkajú statické metódy **Monitor.Enter** a **Monitor.Exit**, môžeme dosiahnuť aj použitím zabudovaného príkazu **lock** jazyka C# 3.0. Definície metód pracovných vlákien by sa použitím tohto príkazu zmenili nasledujúcim spôsobom (aplikované zmeny sú vyznačené sivou podkladovou farbou):

```
// Definícia metódy 1. pracovného vlákna.
static void Inkrementovať()
{
    for (int i = 0; i < 5; i++)
    {
        // Explicitné použitie príkazu lock garantuje
        // exkluzívny prístup k zdieľanému objektu.
        lock(pole)
        {
            pole[i]++;
        }
        Console.WriteLine("[PV1-Inkrementácia]: " +
            "{0}. prvok poľa: {1}.", i + 1, pole[i]);
        Thread.Sleep(500);
    }
}

// Definícia metódy 2. pracovného vlákna.
static void Dekrementovať()
{
    for (int i = 0; i < 5; i++)
```

```
    {
        // Explicitné použitie príkazu lock garantuje
        // exkluzívny prístup k zdieľanému objektu.
        lock(pole)
        {
            pole[i]--;
        }
        Console.WriteLine("[PV2-Dekrementácia]: " +
            "{0}. prvok poľa: {1}.", i + 1, pole[i]);
        Thread.Sleep(500);
    }
}
```

Príkaz **lock** zavádza vyššiu úroveň abstrakcie synchronizačnej operácie. Je však dôležité uvedomiť si, že synchronizácia je realizovaná monitorom, ktorý slúži na uzamknutie objektu počas trvania kritickej sekcie. Výhodou príkazu **lock** voči explicitnému volaniu metód **Monitor.Enter** a **Monitor.Exit** je automatické uvoľnenie zámku nad synchronizovaným objektom v príhodnom momente (to sa deje pri spracovaní uzatváracej zloženej zátvorky, ktorá ohraničuje oblasť platnosti kritickej sekcie).

Nasledujúci fragment zdrojového kódu jazyka C# 3.0 predvádza vláknovo bezpečné vykonávanie transakcií s bankovým účtom:

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

namespace diz
{
    // Deklarácia triedy BankovýÚčet.
    class BankovýÚčet
    {
        // Definície dátových členov triedy.
        private int stav, minimálnyZostatok;
        // Definícia parametrického konštruktura.
        public BankovýÚčet(int počiatočnýVklad, int minimálnyZostatok)
        {
            stav = 0;
            stav += počiatočnýVklad;
            this.minimálnyZostatok = minimálnyZostatok;
```

```
    }
    // Definícia metódy pre vloženie peňazí na účet.
    public void Vložiť(int suma)
    {
        stav += suma;
        Console.WriteLine("Po vložení sumy {0} USD je na " +
            "účte {1} USD.", suma, stav);
    }
    // Definícia metódy pre uhradenie peňazí z účtu.
    public void Uhradiť(int suma)
    {
        if(stav - suma < minimálnyZostatok)
        {
            Console.WriteLine("Transakcia nebola uskutočnená " +
                "z dôvodu prekročenia minimálneho zostatku " +
                "na účte.");
        }
        else
        {
            stav -= suma;
            Console.WriteLine("Po úhrade {0} USD " +
                "je na účte {1} USD.",
                suma, stav);
        }
    }
}

class Program
{
    // Založenie bankového účtu.
    static BankovýÚčet môjÚčet = new BankovýÚčet(1000, 100);
    static void Main(string[] args)
    {
        Thread vláknoA, vláknoB;
        // Vytvorenie dvoch pracovných vlákien.
        vláknoA = new Thread(new ThreadStart(RealizovaťVklady));
        vláknoB = new Thread(new ThreadStart(RealizovaťÚhrady));
        // Spustenie pracovných vlákien.
        vláknoA.Start();
        vláknoB.Start();
        // Čakanie na návrat pracovných vlákien.
        vláknoA.Join();
        vláknoB.Join();
        Console.Read();
    }
    // Metóda 1. pracovného vlákna uskutočňuje kolekciu vkladov
    // peňazí na bankový účet.
    static void RealizovaťVklady()
    {
```

```
            for (int i = 1; i <= 5; i++)
            {
                // Vykonanie vláknovo bezpečnej operácie
                // s bankovým účtom.
                lock (môjÚčet)
                {
                    môjÚčet.Vložiť(200);
                }
                Thread.Sleep(200);
            }
        }

        // Metóda 2. pracovného vlákna uskutočňuje kolekciu úhrad
        // peňazí z bankového účtu.
        static void RealizovaťÚhrady()
        {
            for (int i = 1; i <= 5; i++)
            {
                // Vykonanie vláknovo bezpečnej operácie
                // s bankovým účtom.
                lock (môjÚčet)
                {
                    môjÚčet.Uhradiť(400);
                }
                Thread.Sleep(200);
            }
        }
    }
}
```

Komentár k zdrojovému kódu: Máme jeden bankový účet, s ktorým vykonávame pomocou dvoch pracovných vlákien transakcie. Transakcie sa delia do dvoch kolekcií: v jednej vkladáme na účet peniaze a v druhej zase peňažné sumy posielame na iné účty. Aby každé z pracovných vlákien vykonalo zamýšľanú operáciu s bankovým účtom bezpečným spôsobom, uzamkneme ho pomocou príkazu **lock**. Zámok uvoľňujeme v okamihu, keď je transakcia uskutočnená.

14.3 Uviaznutie vlákien

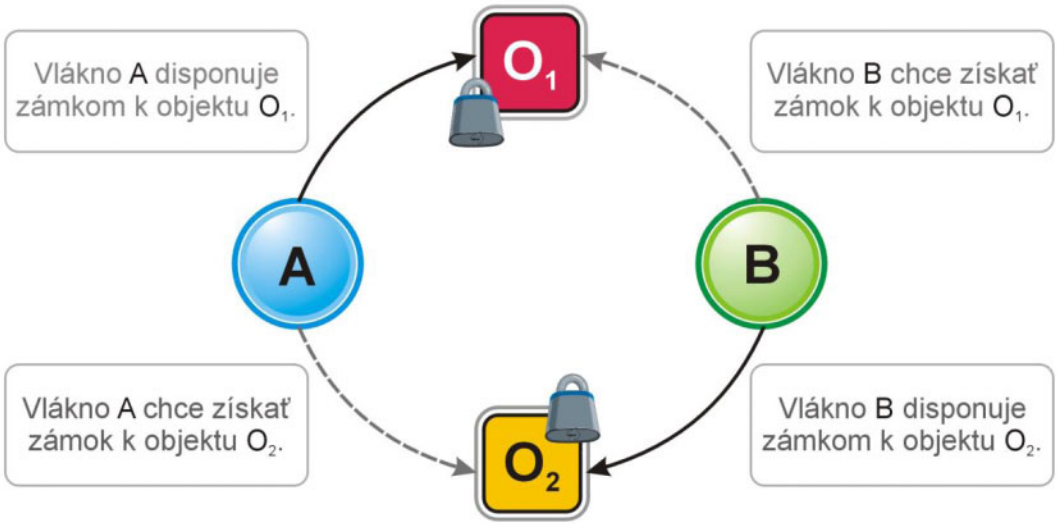
Ešte závažnejšie praktické implikácie ako preteky vlákien majú uviaznutia. Uviaznutím označujeme stav, kedy viaceré programové vlákna disponujú

exkluzívnymi prístupmi k rôznym zdieľaným prostriedkom, pričom určité vlákna sa snažia získať prístup k ďalším zdieľaným prostriedkom, ku ktorým však už exkluzívne pristupujú iné vlákna. Jedna množina vlákien teda čaká, kým nezíska prístup k súprave zdieľaných prostriedkov, ktoré vlastní druhá množina vlákien, zatiaľ čo druhá množina vlákien čaká, kým nezíska prístup k súprave zdieľaných prostriedkov, ktoré vlastní prvá množina vlákien.

Uveďme príklad: Vlákno **A** disponuje exkluzívnym prístupom k zdieľanému prostriedku $\mathbf{O_1}$. Vlákno **B** disponuje exkluzívnym prístupom k zdieľanému prostriedku $\mathbf{O_2}$. Ak bude chcieť vlákno **A** získať prístup k zdieľanému prostriedku $\mathbf{O_2}$ a vlákno **B** bude chcieť získať prístup k zdieľanému prostriedku $\mathbf{O_1}$, tak sa obe vlákna dostávajú do stavu uviaznutia. Je to spôsobené z nasledujúcich dôvodov:

- Vlákno **A** musí čakať, kým nezíska prístup k zdieľanému prostriedku $\mathbf{O_2}$. Vlákno **A** teda zotrváva v nečinnom stave dovtedy, pokiaľ vlákno **B** neuvoľní prístup k zdieľanému prostriedku $\mathbf{O_2}$. Bohužiaľ, nevieme predpovedať kedy a či vôbec vlákno **B** uvoľní prístup k zdieľanému prostriedku $\mathbf{O_2}$.

- Vlákno **B** musí čakať, kým nezíska prístup k zdieľanému prostriedku $\mathbf{O_1}$. Vlákno **B** teda zotrváva v nečinnom stave dovtedy, pokiaľ vlákno **A** neuvoľní prístup k zdieľanému prostriedku $\mathbf{O_1}$. Nanešťastie, nevieme predpovedať kedy a či vôbec vlákno **A** uvoľní prístup k zdieľanému prostriedku $\mathbf{O_1}$.



Obr. 14.4: Uviaznutie dvoch programových vlákien

Uviaznutie zapríčiňuje, že obe vlákna sú zablokované. To znamená, že viacvláknová aplikácia nemôže uskutočniť žiadny progres v realizácii svojich výpočtových procesov. Postup zaznamenáme až vtedy, keď vlákno **A** uvoľní prístup k zdieľanému prostriedku $\mathbf{O_1}$ a vlákno **B** uvoľní prístup k zdieľanému prostriedku $\mathbf{O_2}$.

Uviaznutie vlákien simulujeme v nasledujúcom zdrojovom kóde jazyka C# 3.0:

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

namespace diz
{
   // Deklarácia pomocnej triedy s jedným verejne prístupným
   // dátovým členom.
    class O
    {
        public int x;
        public O(int i)
        {
            x = i;
```

```
    }
}
class Program
{
    // Založenie dvoch statických objektov.
    static O o1 = new O(60);
    static O o2 = new O(40);
    static void Main(string[] args)
    {
        // Vytvorenie dvoch pracovných vlákien.
        Thread vláknoA = new Thread(new ThreadStart(m1));
        Thread vláknoB = new Thread(new ThreadStart(m2));
        // Pomenovanie pracovných vlákien.
        vláknoA.Name = "VláknoA";
        vláknoB.Name = "VláknoB";
        // Spustenie pracovných vlákien.
        vláknoA.Start();
        vláknoB.Start();
        // Čakanie na návrat pracovných vlákien.
        vláknoA.Join();
        vláknoB.Join();
        // Zobrazenie finálnych stavov statických objektov.
        Console.WriteLine("o1.x = {0}", o1.x);
        Console.WriteLine("o2.x = {0}", o2.x);
        Console.Read();

    }
    // Metóda pracovného vlákna A.
    static void m1()
    {
        // Uzamknutie statického objektu o1.
        Monitor.Enter(o1);
        Console.WriteLine("Vlákno A uzamklo objekt o1.");
        Thread.Sleep(500);
        try
        {
            // Zmena hodnoty dátového člena statického objektu o1.
            o1.x = 100;
            // Uzamknutie statického objektu o2.
            Monitor.Enter(o2);
            Console.WriteLine("Vlákno A uzamklo objekt o2.");
            try
            {
                // Zmena hodnoty dátového člena
                // statického objektu o2.
                o2.x = 200;
            }
            finally
            {
```

```
                    // Odomknutie statického objektu o2.
                    Monitor.Exit(o2);
                    Console.WriteLine("Vlákno A odomklo objekt o2.");
                }
            }
            finally
            {
                // Odomknutie statického objektu o1.
                Monitor.Exit(o1);
                Console.WriteLine("Vlákno A odomklo objekt o1.");
            }
        }
        // Metóda pracovného vlákna B.
        static void m2()
        {
            // Uzamknutie statického objektu o2.
            Monitor.Enter(o2);
            Console.WriteLine("Vlákno B uzamklo objekt o2.");
            Thread.Sleep(500);u
            try
            {
                // Zmena hodnoty dátového člena statického objektu o2.
                o2.x = 300;
                // Uzamknutie statického objektu o1.
                Monitor.Enter(o1);
                Console.WriteLine("Vlákno B uzamklo objekt o1.");
                try
                {
                    // Zmena hodnoty dátového člena
                    // statického objektu o1.
                    o1.x = 400;
                }
                finally
                {
                    // Odomknutie statického objektu o1.
                    Monitor.Exit(o1);
                    Console.WriteLine("Vlákno B odomklo objekt o1.");
                }
            }
            finally
            {
                // Odomknutie statického objektu o2.
                Monitor.Exit(o2);
                Console.WriteLine("Vlákno B odomklo objekt o2.");
            }
        }
    }
}
```

Komentár k zdrojovému kódu: Pri analýze uvedeného paralelného programu majme prosím na pamäti, že ide o čo možno najjednoduchšiu riadenú simuláciu uviaznutia pracovných programových vlákien. K uviaznutiu samozrejme dôjde, pretože sú splnené podmienky, ktoré vznik tejto programovej chyby podmieňujú. Pracovnému vláknu **A** sa preto nikdy nepodarí získať zámok nad statickým objektom **o2**. Analogicky, pracovné vlákno **B** nebude nikdy schopné uzamknúť statický objekt **o1**. Informačné správy, ktoré sprevádzajú zmeny stavov statických objektov teda používateľ tohto paralelného programu ani raz neuvidí. Z pohľadu používateľa sa bude program javiť ako nečinný, čo je pochopiteľne dôsledok súperenia pracovných vlákien, ktoré nemá víťaza.

Uviaznutie vlákien môžeme vyriešiť niekoľkými spôsobmi, z ktorých uvedieme nasledujúce dva:

1. **Privatizácia objektov.** Podstatou techniky privatizácie objektov je poskytnúť každému pracovnému vláknu vlastnú kópiu pôvodne zdieľaného objektu. Keď bude každé pracovné vlákno vykonávať operácie s vlastným objektom, nemôže dôjsť k chybe spôsobenej nesynchronizovaným konkurenčným prístupom.

2. **Determinácia identického poradia získavania exkluzívnych prístupov k objektom.** Ak stanovíme presné poradie, v akom budú pracovné vlákna pristupovať k zdieľaným objektom, vyhneme sa riziku ich uviaznutia. Napríklad, v našom príklade by bolo vhodné, aby najskôr manipulácie so statickými objektmi vykonalo pracovné vlákno **A** a až potom pracovné vlákno **B**. V záujme minimalizácie času, počas ktorého vlákno **B** zotrváva v nečinnom stave, môžeme toto vlákno zaťažiť realizáciou iných operácií. Vlákno **B** sa dostane k statickým objektom vo chvíli, keď vlákno **A** tieto objekty odomkne, a tak sprístupní.

14.4 Atomické operácie

Pri programovaní viacvláknových aplikácií v jazyku C# 3.0 môžu vývojári využiť rad synchronizačných objektov, ku ktorým patria atomické operácie, monitory, mutexy a synchronizačné udalosti.

Atomickú operáciu definujeme ako množinu parciálnych operácií A_T, ktorá je vždy spracovaná ako diskrétna jednotka, bez rizika prerušenia svojej exekúcie.

$$A_T = \left\{at_{op_1}, at_{op_2}, \dots, at_{op_n}\right\}, n \in N$$

Pritom platí, že parciálne operácie $at_{op_{1\dots n}}$ sú spracované všetky v rámci jednej exekučnej relácie. Ak sú všetky parciálne operácie $at_{op_{1\dots n}}$ úspešne realizované, atomická operácia reprezentovaná množinou A_T bola úspešne aplikovaná. Ak sa však stane, že jedna parciálna operácia $(at_{op_i}, \forall i: 1 \leq i \leq n)$ nebude môcť byť vykonaná, zmeny uskutočnené predchádzajúcimi parciálnymi operáciami budú anulované. V tomto prípade vravíme, že atomická operácia nebola úspešne aplikovaná.

Pri programovaní paralelných aplikácií v jazyku C# 3.0 využijeme atomické operácie predovšetkým na vláknovo bezpečnú modifikáciu hodnôt premenných hodnotových a odkazových dátových typov. To je dôležité, pretože napríklad inkrementácia hodnoty premennej nie je implicitne atomickou operáciou. V skutočnosti sa táto operácia skladá z troch krokov:

1. Načítanie hodnoty premennej z operačnej pamäte do registra exekučného jadra viacjadrového procesora.

2. Inkrementácia načítanej hodnoty.

3. Uloženie inkrementovanej hodnoty z registra späť do premennej nachádzajúcej sa v operačnej pamäti.

Vlákno **A**, ktoré bude vykonávať túto triádu operácií, môže byť kedykoľvek prerušené s tým, že spôsobí porušenie dátovej integrity cieľovej premennej **P**. Ak vlákno **A** úspešne vykoná prvé dva kroky triády, pričom dôjde k jeho prerušeniu, inkrementačná operácia nebude vykonaná v celku. Predstavme si, že vlákno **B** zmení počas trvania svojho časového kvanta hodnotu premennej **P**. Keď znova príde na rad vlákno **A**, bude sa snažiť dokončiť zahájenú inkrementáciu pôvodnej hodnoty premennej **P**. Vlákno **A** teda uloží do premennej **P** inkrementovanú hodnotu, čím prepíše hodnotu, ktorú do tejto premennej uložilo vlákno **B**. Stav premennej **P** nie je platný, pretože s premennou boli uskutočnené dve nesynchronizované zápisové operácie.

Základné atomické operácie s premennými nám v programovacom jazyku C# 3.0 dovoľuje uskutočňovať statická trieda **Interlocked** situovaná v mennom priestore **System.Threading**.

Trieda **Interlocked** definuje tieto významné statické metódy:

1. Metóda **Increment** realizuje bezpečnú atomickú inkrementáciu 32- alebo 64-bitovej hodnoty cieľovej premennej integrálneho hodnotového dátového typu.

2. Metóda **Decrement** realizuje bezpečnú atomickú dekrementáciu 32- alebo 64-bitovej hodnoty cieľovej premennej integrálneho hodnotového dátového typu.

3. Metóda **Exchange** realizuje bezpečnú atomickú operáciu priradenia hodnoty do 32- alebo 64-bitovej cieľovej premennej integrálneho hodnotového, reálneho hodnotového, alebo odkazového dátového typu.

Definície parametrických statických metód **Increment** a **Decrement** triedy **Interlocked** majú takúto generickú podobu:

```
public static class Interlocked
{
    public static T Increment(ref T premenná)
    {
        // Telo metódy je vynechané.
    }

    public static T Decrement(ref T premenná)
    {
        // Telo metódy je vynechané.
    }
}
```

kde:

- **T** je integrálny celočíselný dátový typ **int** alebo **long**.
- **ref** je kľúčové slovo určujúce, že argument bude formálnemu parametru metódy odovzdaný odkazom.
- **premenná** je identifikátor premennej, ktorej hodnota je inkrementovaná alebo dekrementovaná príslušnou statickou metódou.

Praktická aplikácia atomických operácií je prostredníctvom spomenutých metód triedy **Interlocked** vskutku intuitívna:

```
class Buffer
{
    private int počítadloReferencií;
    public Buffer()
    {
        this.počítadloReferencií = 0;
    }
    public void ZískaťReferenciu()
    {
        // Inkrementačná atomická operácia.
        Interlocked.Increment(ref this.počítadloReferencií);
    }
    public void UvoľniťReferenciu()
    {
```

```
        // Dekrementačná atomická operácia.
        Interlocked.Decrement(ref this.počítadloReferencií);
    }
    public int ZistiťPočetReferencií()
    {
        return this.počítadloReferencií;
    }
}
```

Komentár k zdrojovému kódu: Typickým príkladom použitia atomických inkrementačno-dekrementačných operácií je počítanie referencií, ktoré sú naviazané na objekt istej triedy. V našej ukážke deklarujeme triedu **Buffer**, v tele ktorej definujeme metódy na získanie a uvoľnenie referencií. Objekt triedy obsahuje interné počítadlo referencií, ktoré reflektuje aktuálny počet naviazaných referencií. Pri aktivácii inštančnej metódy **ZískaťReferenciu** voláme statickú metódu **Increment** triedy **Interlocked**, čím bezpečným spôsobom zvyšujeme hodnotu interného počítadla referencií.

Analogicky, pri volaní inštančnej metódy **UvoľniťReferenciu** zahajujeme prostredníctvom statickej metódy **Interlocked.Decrement** vláknovo bezpečné zníženie hodnoty interného počítadla.

Ak inkrementáciu a dekrementáciu implementujeme ako atomické operácie, máme záruku ich kompletného spracovania. To je veľká konkurenčná výhoda najmä v situácii, keď viacero programových vlákien súčasne manipuluje s jedným objektom triedy **Buffer**. Ak by sme za týchto okolností nezapracovali atomické operácie, niektoré z vlákien by mohli zachytiť objekt v nekonzistentnom stave (teda v stave, ktorý by nezodpovedal skutočnému počtu referencií determinovanému interným počítadlom).

Predostretú deklaráciu triedy **Buffer** by sme mohli pre potreby praktického nasadenia ďalej rozšíriť. Napríklad tak, že by sme načítavali kvantá dát až po určitú hranicu (determinovanú počtom referencií), a potom by sme celý obsah buffera uložili do fyzického súboru na pevnom disku či na inom záznamovom médiu (táto akcia sa často označuje ako vyprázdnenie buffera). Vyprázdnenie buffera by

implikovalo uvoľnenie všetkých naviazaných referencií s opätovným nastavením interného počítadla na nulovú hodnotu.

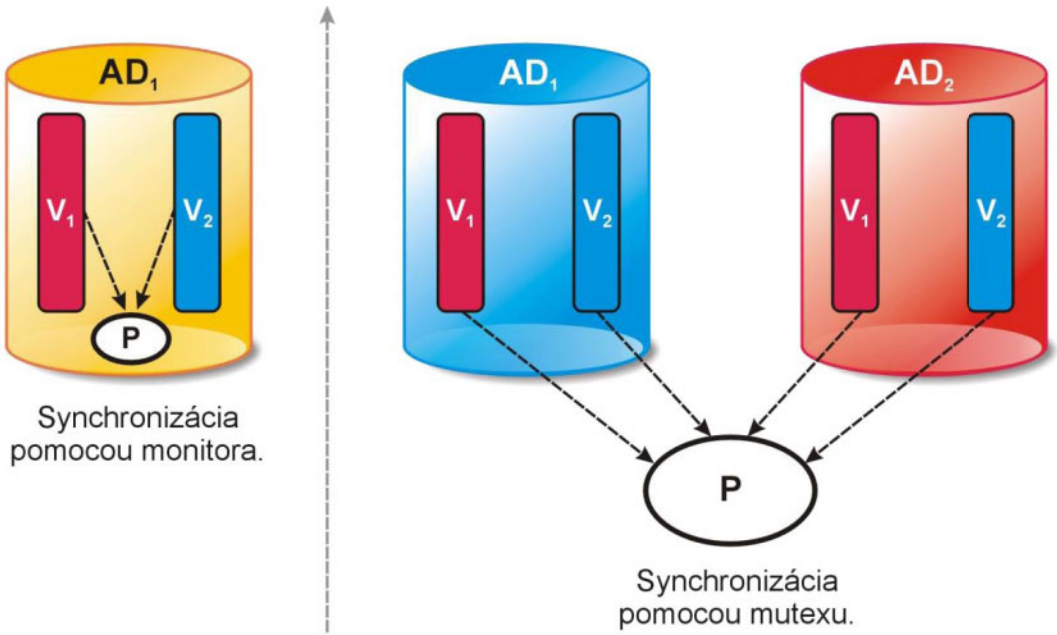
Ak to je možné, snažíme sa uprednostňovať atomické operácie pred inými synchronizačnými primitívami, predovšetkým pred monitormi a mutexmi.

14.5 Mutexy

Mutex[26] je synchronizačný objekt, ktorý sa používa podobne ako monitor, avšak jeho pole pôsobnosti je omnoho širšie. Zatiaľ čo monitor nie je systémovým synchronizačným objektom, mutex takýmto objektom je. Riadená implementácia monitora, ktorú ponúka bázová knižnica vývojovo-exekučnej platformy Microsoft .NET Framework 3.5, pôsobí ako nadstavba nad vrstvou systémových synchronizačných objektov.

Ako sme už uviedli, monitor je efektívnym riešením pri synchronizácii prístupu viacerých programových vlákien k zdieľanému prostriedku, ktorý existuje v jednej aplikačnej doméne, resp. v jednom fyzickom procese riadenej aplikácie. V tomto scenári nasadenia preukazuje monitor optimálne výkonnostné charakteristiky. Na druhej strane, niekedy je nutné synchronizovať prístup k zdieľanému prostriedku z viacerých aplikačných domén, resp. z viacerých fyzických procesov riadených aplikácií súčasne. Keďže monitor nedokáže prekračovať hranice aplikačných domén a fyzických procesov, je nutné za týmto účelom použiť mutex. Mutex je hlavným synchronizačným objektom, ktorý nachádza svoje uplatnenie pri uskutočňovaní interprocesových synchronizácií. Porovnanie nasadenia monitora a mutexu vizualizuje obr. 14.5.

[26] Názov „mutex“ vznikol skrátením viacslovného termínu „mutually exclusive“, teda „navzájom sa vylučujúci“. To je príznačné, pretože mutex chráni zdieľaný prostriedok v rámci kritickej sekcie, takže k tomuto prostriedku smie počas trvania kritickej sekcie pristupovať práve jedno programové vlákno.



Obr. 14.5: Porovnanie využitia monitora a mutexu
ako dvoch synchronizačných objektov

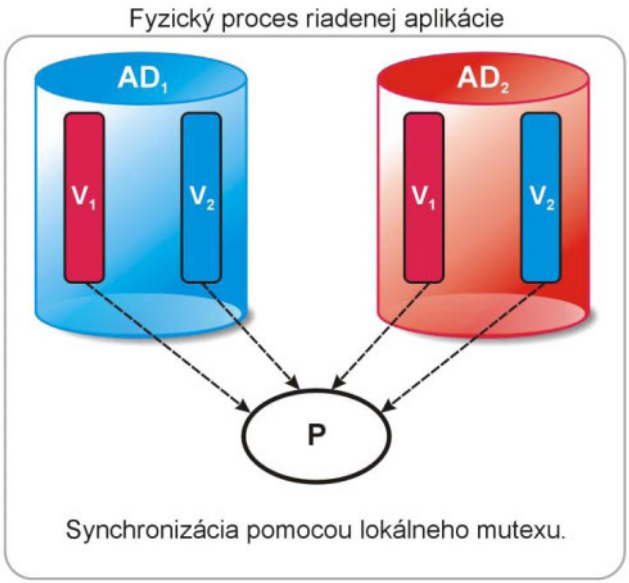
Komentár k obr. 14.5: Najskôr vysvetlíme synchronizáciu pomocou monitora. V ľavej časti obrázka vidíme aplikačnú doménu s identifikátorom **AD_1**. Táto aplikačná doména obsahuje 2 programové vlákna s identifikátormi **v_1** a **v_2**. Obe pracovné vlákna pristupujú k zdieľanému prostriedku s identifikátorom **P**. Synchronizácia pomocou monitora umožňuje korektnú funkčnosť riadenej aplikácie a eliminuje vznik kolíznych stavov medzi pracovnými vláknami. Teraz budeme pokračovať ozrejmením mutexu. V pravej časti obrázka je znázornená riadená aplikácia s 2 aplikačnými doménami (**AD_1** a **AD_2**). Každá z aplikačných domén obsahuje dvojicu pracovných vlákien (**v_1** a **v_2**). Keďže obe aplikačné domény patria do jedného fyzického procesu riadenej aplikácie, môžu pristupovať k zdieľanému prostriedku **P**. Synchronizačné akcie zabezpečí v tomto kontexte mutex, ako synchronizačný objekt, ktorý smie byť použitý naprieč rôznymi aplikačnými doménami.

Mutexy delíme do nasledujúcich skupín:

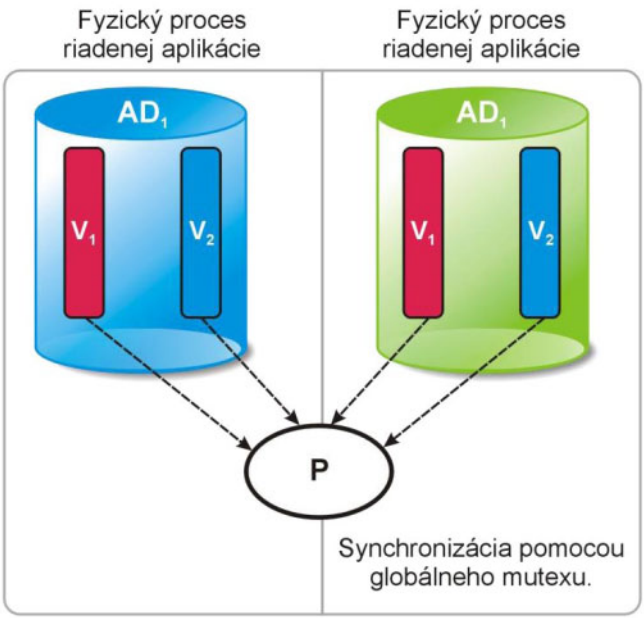
1. **Lokálne mutexy.** Lokálne mutexy môžu byť použité na synchronizáciu prístupu viacerých programových vlákien k zdieľanému prostriedku, pričom tieto vlákna sídlia buď v totožnej aplikačnej doméne fyzického procesu riadenej aplikácie, alebo v rôznych aplikačných doménach fyzického procesu riadenej aplikácie. Lokálne mutexy sú označované tiež ako nepomenované, resp. anonymné mutexy. Lokálne mutexy podporujú intraprocesovú synchronizáciu.

2. **Globálne mutexy.** Globálne mutexy slúžia na synchronizáciu prístupu viacerých programových vlákien k zdieľanému prostriedku, pričom tieto vlákna sídlia v rôznych aplikačných doménach rôznych fyzických procesov riadených aplikácií. Z uvedeného vyplýva, že globálne mutexy sú objektmi, ktoré podporujú interprocesovú synchronizáciu. Keďže globálne mutexy vytvorené v rôznych fyzických procesoch riadených aplikácií musia mať rovnaké identifikátory, sú tieto mutexy často nazývané pomenované mutexy, resp. systémové mutexy.

Rozdiely v použití lokálnych a globálnych mutexov uvádzame na obr. 14.6 – 14.7.



Obr. 14.6: Synchronizácia pomocou lokálneho mutexu



Obr. 14.7: Synchronizácia pomocou globálneho mutexu

Lokálny mutex môžeme využiť napríklad pri návrhu bezpečného zásobníka[27]:

```
class BezpečnýZásobník
{
    private Stack zásobník;
    private Mutex mutex;

    public BezpečnýZásobník()
    {
        zásobník = new Stack();
        mutex = new Mutex();
    }
    public void Vložiť(object objekt)
    {
        // Získanie mutexu k zásobníku.
        mutex.WaitOne();
        try
        {
            // Vloženie objektu do chráneného zásobníka.
            zásobník.Push(objekt);
        }
        finally
        {
            // Uvoľnenie mutexu zo zásobníka.
            mutex.ReleaseMutex();
        }
    }
    public object Vybrať()
    {
        object objekt;
        // Získanie mutexu k zásobníku.
        mutex.WaitOne();
        try
        {
            if (zásobník.Count > 0)
            {
                // Vyňatie objektu z chráneného zásobníka.
                objekt = zásobník.Pop();
            }
            else
            {
                throw new Exception("Zásobník je prázdny.");
            }
        }
```

[27] Fragment zdrojového kódu predpokladá, že sú zavedené menné priestory **System.Collections** a **System.Threading**.

```
        finally
        {
            // Uvoľnenie mutexu zo zásobníka.
            mutex.ReleaseMutex();
        }
        return objekt;
    }
}
```

Komentár k zdrojovému kódu: Trieda **BezpečnýZásobník**, ktorú sme vytvorili, predstavuje vláknovo bezpečnú obálku vstavanej zásobníkovej a negenerickej triedy **Stack** z menného priestoru **System.Collections**. Zásobník definuje dve základné metódy: **Vložiť** na uloženie nového objektu do zásobníka a **Vybrať** na vyňatie posledne vloženého objektu zo zásobníka (zásobník pracuje podľa organizačného modelu LIFO). Bezpečný zásobník je schopný synchronizovaným spôsobom pridávať a vyberať objekty, a to vďaka začleneniu mutexu ako hlavného synchronizačného objektu. Definícia metódy **Vložiť** ukazuje, ako s mutexom pracujeme.

Aby sme získali mutex, voláme inštančnú metódu **WaitOne** v súvislosti s objektom triedy **Mutex**. V našom prípade využívame bezparametrickú verziu metódy **WaitOne**, čím vyjadrujeme ochotu čakať variabilne dlhý časový interval, kým nezískame mutex. Podotknime, že metóda **WaitOne** je preťažená, pričom ďalšie definície nám umožňujú explicitne nastaviť maximálnu časovú periódu, ktorú si môžeme dovoliť alokovať v záujme získania mutexu. V tomto kontexte získame lokálny mutex, ktorý smieme použiť na synchronizáciu manipulačnej operácie so zásobníkom. Len čo je objekt bezpečne uložený do zásobníka, uvoľňujeme pridelený mutex volaním metódy **ReleaseMutex**. Všimnime si, že dealokácia mutexu je umiestnená v bloku **finally**. Tým eliminujeme riziko neuvoľnenia raz prideleného mutexu (mutex, ktorý nie je vo vhodnej chvíli uvoľnený, sa stáva opusteným mutexom).

Analogicky postupujeme aj pri použití mutexu v súvislosti s vybratím objektu zo zásobníka. Samozrejme, objekt môžeme zo zásobníka získať len vtedy, pokiaľ nie je zásobník prázdny. Ak príde požiadavka na vyňatie objektu z prázdneho zásobníka, generujeme chybovú výnimku.

15 Varianty paralelizmu

Počítačové vedy definujú niekoľko variantov paralelizmu, ktoré sú zvyčajne asociované s konkrétnymi paralelnými výpočtovými modelmi.

Podľa úrovne zapojenia ľudského faktora do procesu paralelizácie počítačovej aplikácie vymedzujeme **implicitný** a **explicitný** paralelizmus.

Implicitný paralelizmus znamená automatickú paralelizáciu programu bez intervencie vývojára. V rámci tohto modelu je program vytváraný na sekvenčnej báze, pričom detekciu potenciálne paralelizovateľných segmentov a ich následnú implicitnú paralelizáciu vykoná automaticky prekladač. Pri implementácii implicitného paralelizmu nie sú potrebné žiadne direktívy, značky či špeciálne funkcie, ktoré by programátor musel v záujme paralelizácie vkladať priamo do zdrojového kódu. Celková zodpovednosť za paralelizáciu programu zostáva na prekladači, ktorý sa bude pomocou sofistikovaných techník analýzy zdrojového kódu snažiť realizovať transformáciu exekúcie vymedzenej množiny programových príkazov zo sekvenčnej na paralelnú.

Explicitný paralelizmus vyžaduje kooperáciu človeka, v rámci ktorej vývojár najskôr identifikuje tie partie zdrojového kódu počítačovej aplikácie, ktoré budú spracúvané súbežne a následne napíše zdrojový kód, ktorý bude požadovanú paralelizáciu uskutočňovať. Explicitný paralelizmus smie byť realizovaný v rôznych úrovniach abstrakcie. Pritom platí, že vyššia úroveň abstrakcie explicitného paralelizmu prináša tieto efekty:

1. Znižuje mieru zainteresovanosti vývojára v procese paralelizácie softvéru.
2. Zvyšuje mieru koncentrácie vývojára na riešenie úlohy.
3. Eliminuje nutnosť disponovať špecifickými technickými znalosťami o nízkoúrovňovej paralelnej implementácii riešenej úlohy.
4. Zvyšuje pracovnú produktivitu vývojára.
5. Skracuje čas potrebný na vývoj škálovateľnej počítačovej aplikácie, ktorá dokáže flexibilne využiť všetku výpočtovú kapacitu počítačového systému.

Nižšia abstraktná úroveň explicitného paralelizmu sa vyznačuje tvorbou zdrojového kódu, ktorý preberá kompetencie za realizáciu týchto činností:

1. Explicitná tvorba programových vlákien.
2. Selekcia a delegovanie úloh, ktoré budú programové vlákna vykonávať.
3. Eliminácia výskytu kolíznych stavov s využitím synchronizačných primitív.
4. Celkový manažment programových vlákien, ktorý spočíva v optimálnom riadení ich životných cyklov.

Ak sa rozhodneme implementovať nižšiu abstraktnú úroveň explicitného paralelizmu, budeme musieť zvládnuť všetky spomenuté aktivity. Tým síce získavame úplnú kontrolu nad správou programových vlákien, no vystavujeme sa potenciálnemu riziku vzniku veľmi nebezpečných chýb nedeterministického charakteru. Tie majú tendenciu objavovať sa najmä pri zložitých programoch s rozsiahlou základňou zdrojového kódu a pri programoch, ktoré sú napojené na externé knižničné moduly s nejasnou, resp. nedokumentovanou funkcionalitou. Keďže všetky nízkoúrovňové činnosti súvisiace s riadením životných cyklov programových vlákien musíme explicitne naprogramovať, so zvyšujúcou náročnosťou programátorskej práce sa znižuje jej produktivita.

Stredná úroveň abstrakcie explicitného paralelizmu je spojená s využitím špeciálnych direktív, kľúčových slov, modifikátorov a funkcií na podporu paralelizácie. Pri implementácii explicitného paralelizmu so strednou úrovňou abstrakcie stačí, keď programátor prepracuje bloky zdrojového kódu aplikácie, ktoré má záujem paralelizovať. Stredná úroveň abstrakcie explicitného paralelizmu môže byť realizovaná rôznymi spôsobmi:

1. Pomocou programových konštrukcií, ktoré sú priamo vstavané do jazykovej špecifikácie použitého programovacieho jazyka. Tieto programovacie jazyky boli už navrhnuté s myšlienkou priamej podpory paralelizácie programových činností.

2. Pomocou aditívnych programových konštrukcií, ktoré rozširujú pôvodnú jazykovú špecifikáciu použitého programovacieho jazyka. Hoci sa pri vytváraní programových jazykov tejto kategórie nepočítalo s priamou podporou paralelizácie, ich tvorcovia túto podporu začlenili do syntakticko-sémantických rozšírení, ktoré boli začlenené aj do prekladačov týchto jazykov.

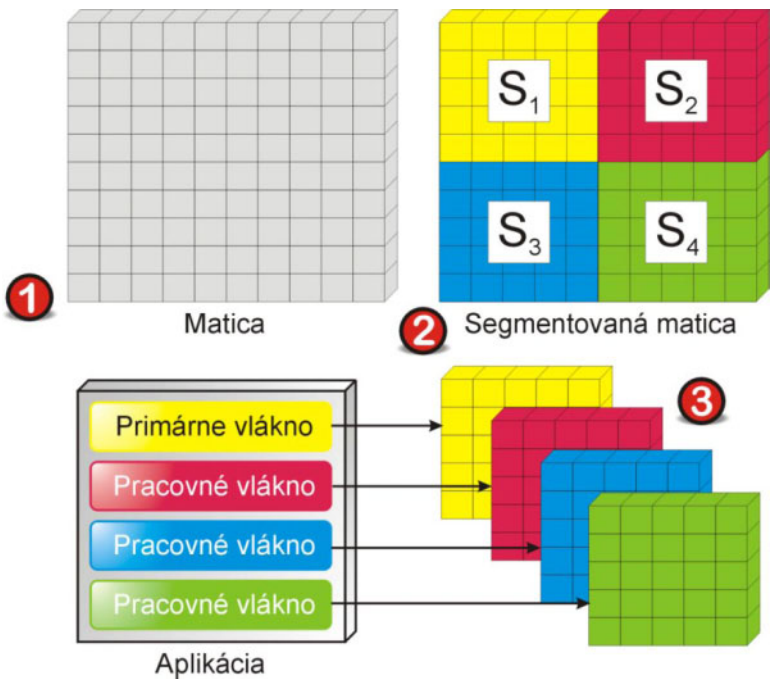
3. Pomocou externých knižničných modulov, ktoré poskytujú aplikačné programové rozhranie na podporu paralelizácie procesov (tzv. paralelné rozhrania API). Externé knižničné moduly poskytujú prostriedky na vytváranie programových vlákien a riadenie ich životných cyklov.

Vysokú úroveň abstrakcie explicitného paralelizmu asociujeme s použitím platforiem na podporu paralelizácie programových činností. Paralelné platformy majú formu vysoko abstraktných aplikačných programových rozhraní. Tieto rozhrania API nám umožňujú pracovať nie s programovými vláknami, ale s abstraktnými programovými konštrukciami, prostredníctvom ktorých smieme modelovať úlohy, ktoré majú byť vykonávané paralelne (a rovnako aj vzťahy medzi týmito úlohami). Paralelné platformy dovoľujú vývojárom nahliadať na aplikáciu ako na množinu úloh, ktoré môžu byť realizované súbežne a nie ako na množinu programových vlákien, ktorých životné cykly je potrebné explicitne riadiť.

Podľa dekompozičných techník rozlišujeme nasledujúce varianty paralelizmu:

1. **Dátový paralelizmus.** Dátový paralelizmus znamená súbežné vykonávanie rovnakej činnosti na rôznych inštanciách dátovej štruktúry, alebo súbežné vykonávanie rovnakej činnosti na rôznych dátových blokoch identickej inštancie dátovej štruktúry. Napríklad dátový paralelizmus môžeme využiť pri inicializovaní štvorcovej matice pseudonáhodnými celými číslami. Ak predpokladáme, že n je počet programových vlákien, tak v rámci dekompozície problému navrhneme vláknovú kompozíciu aplikácie tak, aby každé vlákno vykonalo $1/n$ objemu celkovej práce. Teda každé vlákno bude zodpovedné za inicializáciu $1/n$ štvorcovej matice. Pre 4-vláknovú aplikáciu

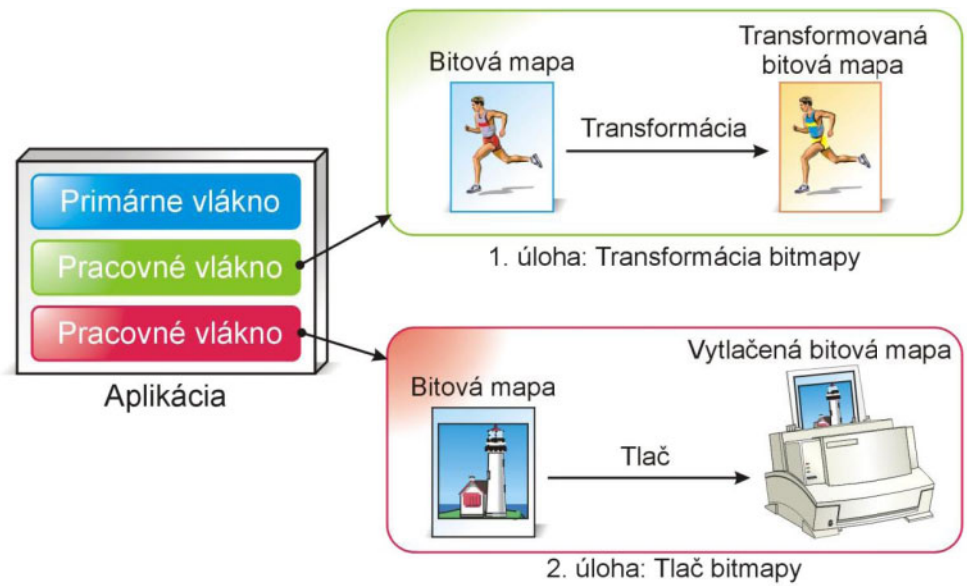
to znamená, že každé vlákno vykoná 1/4, čiže 25 % celkovej práce. Ak bude spracúvaná štvorcová matica typu 10000x10000, tak potrebujeme dovedna inicializovať 100 miliónov prvkov. Pomocou dekompozičnej techniky rozdelíme celú maticu na 4 bloky (segmenty), a nariadime, aby každé z vlákien inicializovalo jeden z týchto blokov. Povedané inak, každé programové vlákno 4-vláknovej aplikácie uskutoční inicializáciu 25 miliónov prvkov. Tak sme schopní dosiahnuť optimálnu distribúciu pracovného zaťaženia naprieč jednotlivými vláknami. Vzhľadom na to, že každé vlákno bude operovať na diskrétnom dátovom bloku, nemusíme synchronizovať ich pracovné modely. Schematické znázornenie postupu prác pri využití dátového paralelizmu prináša obr. 15.1.



Obr. 15.1: Dátový paralelizmus

2. **Úlohový paralelizmus.** O úlohovom paralelizme hovoríme vtedy, ak program vykonáva súbežne viaceré úlohy. Tieto paralelne realizované úlohy

sú aplikované na rôzne inštancie dátových štruktúr, resp. na rôzne inštancie identickej dátovej štruktúry. (Ak by mali byť paralelné úlohy vykonávané na identickej inštancii dátovej štruktúry, tak by sme museli synchronizovať ich pracovné modely.) Uvažujme grafický editor s viacdokumentovým rozhraním (Multiple Document Interface, MDI). Z množiny funkcií grafického editora si vyberieme dve funkcie (f_1 a f_2), ktoré budú môcť byť vykonávané paralelne. Funkciou f_1 bude grafická transformácia (prvej) bitovej mapy. Funkciou f_2 bude tlač (druhej) bitovej mapy, pričom táto operácia sa bude odohrávať počas spracovania funkcie f_1, teda počas realizácie grafickej transformácie. Obe úlohy reprezentované funkciami budú vykonávané paralelne, pretože v jednom okamihu bude editor vykonávať grafickú transformáciu jednej bitovej mapy a súčasne tlačiť inú bitovú mapu. Ak by funkcie f_1 a f_2 neboli vykonávané paralelne, používateľ by nemohol vydať pokyn na tlač bitovej mapy skôr, než by sa dokončila grafická transformácia predchádzajúcej bitovej mapy. Táto skutočnosť by však bola chápaná ako obmedzenie a v konečnom dôsledku by spôsobila zníženie subjektívnej výkonnosti grafického editora v očiach používateľa.



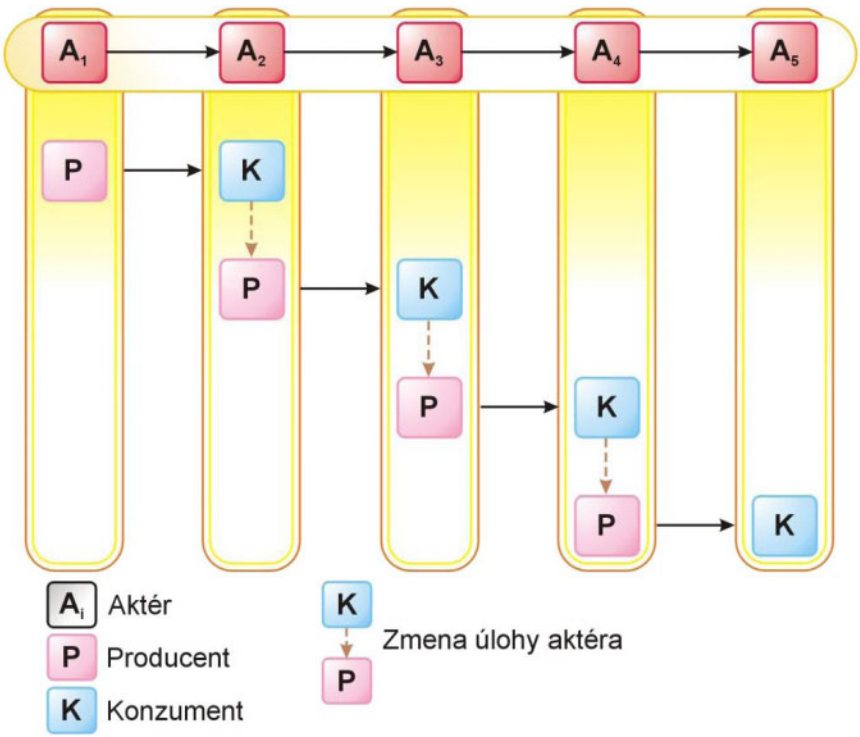
Obr. 15.2: Úlohový paralelizmus

3. **Paralelizmus dátových tokov.** Kým pri dátovom paralelizme sme realizovali totožnú činnosť nad rôznymi blokmi inštancie dátovej štruktúry a pri úlohovom paralelizme sme riadili spracovanie rôznych úloh na rôznych inštanciách dátovej štruktúry, tak pri paralelizme dátových tokov sa zameriavame na analýzu toku dát (zapuzdrených v inštanciách dátovej štruktúry) naprieč viacerými úlohami. Základné princípy paralelizmu dátových tokov môžeme charakterizovať na modeli producent/konzument. V záujme ďalších myšlienkových pochodov prijmime dohovor, že každý zo spomenutých aktérov bude reprezentovaný jedným programovým vláknom.

   Pre model producent/konzument platí, že dáta plynú od prvého aktéra (producenta) k druhému aktérovi (konzumentovi). Pritom medzi oboma aktérmi existuje jasne determinovaná závislosť, ktorá implikuje skutočnosť, že výstup prvého aktéra (producenta) sa stáva vstupom druhého aktéra (konzumenta). To znamená, že vlákno konzumenta môže zahájiť vykonanie požadovanej operácie na dátach až vtedy, keď mu tieto dáta poskytne vlákno producenta. Paralelizmus dátových tokov sa uskutočňuje napríklad pri spracovaní zdrojového kódu programu, ktorý bol napísaný v istom programovacom jazyku. Najskôr musí byť vykonané predspracovanie zdrojového kódu predprocesorom a až potom sa môže k slovu dostať kompilátor. Po preklade zdrojového kódu prichádza na rad spojovací program (linker), ktorý zabezpečí generovanie strojového, resp. pseudostrojového kódu. Tento kód bude nakoniec zaliaty do priamo spustiteľného súboru programu.

   Model producent/konzument, ktorý pracuje s dvomi aktérmi, môžeme podľa potreby modifikovať tak, aby bol schopný pracovať s ľubovoľným počtom aktérov. Označme počet aktérov A_i, pričom $i \in < 1, n >$ a $n \in N$. Potom platí, že pre každé $i < n$ je každý aktér A_i producentom a každý aktér A_{i+1} je konzumentom. Pre $i \geq 2$ sa konzument A_i stáva producentom vo vzťahu k nasledujúcemu aktérovi A_{i+1} (obr. 15.3).

Pracovný model paralelizmu dátových tokov môžeme prirovnať k sériovej výrobe na výrobnom páse. Podobne, ako je výrobný proces segmentovaný na štádiá, v ktorých sa modifikuje aktuálny stav výrobku, tak sa správajú aktéri v reťazci tvorenom z producentov a konzumentov. Len čo jeden aktér vykoná s dátami požadovanú operáciu, dáta sú transportované ďalšiemu aktérovi, ktorý pokračuje v ich spracovaní. Tento kolobeh sa opakuje dovtedy, kým dáta neprejdú celým reťazcom aktérov a nebudú modifikované do svojej finálnej podoby.



Obr. 15.3: Paralelizmus dátových tokov: model producent/konzument

Podľa úrovne implementácie paralelizmu môžeme diferencovať medzi **deklaratívnym** a **imperatívnym** paralelizmom.

Deklaratívny paralelizmus sa vyznačuje vysokou mierou abstrakcie, pretože dovoľuje vývojárom sústrediť sa na splnenie stanoveného cieľa. Pomocou deklaratívneho paralelizmu teda vyjadrujeme náš zámer, čo chceme uskutočniť, no už nás nezaujíma, ako to chceme uskutočniť. Vďaka funkcionálnym programovacím konštrukciám dokáže deklaratívny paralelizmus zaviesť vyššiu mieru abstrakcie, s ktorou následne programátori pracujú.

Imperatívny paralelizmus sa naopak koncentruje na deterministické vykonanie exaktne navrhnutej postupnosti krokov, ktorú je nutné realizovať. V porovnaní s deklaratívnym paralelizmom zavádza imperatívny paralelizmus nižšiu mieru abstrakcie. Ak vývojári pracujú s imperatívnym paralelizmom, musia riešiť všetky implementačné detaily paralelizmu na strednej, prípadne aj nízkej úrovni. Zatiaľ čo deklaratívny paralelizmus nám dovoľuje sústrediť sa na činnosť, ktorú chceme vykonať, imperatívny paralelizmus sa skôr zameriava na to, ako bude táto činnosť uskutočnená.

Deklaratívny a imperatívny prístup k paralelizácii počítačových programov existujú v počítačových vedách niekoľko desaťročí. Oba prístupy formujú samostatné programovacie paradigmy, ktoré môžeme uplatniť pri paralelizácii počítačových aplikácií. V praktických podmienkach však vývojári spravidla dávajú prednosť imperatívnemu paralelizmu, čo je podľa nás spôsobené predovšetkým imperatívnou povahou programovacích jazykov s najväčšou penetráciou na trhu. Napriek tomu si myslíme, že použitie deklaratívnych prostriedkov pri tvorbe paralelných aplikácií je veľmi výhodné, pretože nielenže eliminuje vznik mnohých kolíznych stavov, ale umožňuje vývojárom pracovať s vysoko abstraktnými programovacími konštrukciami. Konkurenčnou výhodou deklaratívneho programovania je bezpochyby možnosť koncentrovať sa na finálne riešenie problému a nie na presný reťazec činností, ktoré k tomuto riešeniu vedú.

Z posledného vývoja udalostí v oblasti tvorby paralelných aplikácií dokážeme identifikovať jav preberania mnohých konštrukcií deklaratívneho programovania a ich následnej implementácie do imperatívnych programovacích jazykov. Týmto spôsobom vznikajú hybridné programovacie jazyky, ktoré majú imperatívne a aj

deklaratívne črty. Prelínanie programovacích paradigiem tak vyúsťuje do synergického efektu, citeľne skracujúceho čas potrebný na analýzu, návrh a implementáciu paralelných počítačových aplikácií.

16 Praktické cvičenie: Paralelizácia sekvenčnej riadenej aplikácie

Komerčných softvérových vývojárov obvykle veľmi interesujú praktické riešenia, ktoré ukazujú, ako dokáže paralelné programovanie zvýšiť výkonnosť doposiaľ rýdzo sekvenčných aplikácií. Preto sa v tejto kapitole sústredíme na proces paralelizácie sekvenčnej riadenej aplikácie. Táto aplikácia beží v konzolovom okne a vykonáva inverziu (grafickú transformáciu) kolekcie bitových máp. Naša aplikácia bola pôvodne vytvorená ako sekvenčná, čo znamená, že spracovanie inverzných operácie prebiehalo synchrónne: najskôr bola uskutočnená inverzia prvej bitovej mapy, potom druhej, tretej, atď., až pokiaľ neboli invertované všetky požadované bitové mapy.

Inverziu ako grafickú transformáciu sme implementovali v samostatnej triede **Bitmapa**, ktorá je deklarovaná v zdrojovom súbore Bitmapa.cs. Syntaktický obraz tohto zdrojového súboru je takýto:

```
using System;
using System.Collections.Generic;
// Import potrebného menného priestoru.
using System.Drawing;
using System.Linq;
using System.Text;

namespace diz
{
    // Deklarácia triedy, ktorá zapuzdruje funkcionalitu na realizáciu
    // inverzie bitových máp.
    class Bitmapa
    {
        private Bitmap bitováMapa;
        private string súbor;
        // Parametrický konštruktor triedy.
        public Bitmapa(string súbor)
        {
            bitováMapa = new Bitmap(súbor);
            this.súbor = súbor;
        }
        // Metóda vykonavajúca grafickú transformáciu bitovej mapy.
        public void Invertovať()
```

```
        {
            int x, y;
            Color farba;
            for (x = 0; x < bitováMapa.Width; x++)
            {
                for(y = 0; y < bitováMapa.Height; y++)
                {
                    farba = bitováMapa.GetPixel(x, y);
                    bitováMapa.SetPixel(x, y,
                        Color.FromArgb(255 - farba.R, 255 - farba.G,
                        255 - farba.B));
                }
            }
            // Uloženie invertovanej bitovej mapy do samostatného súboru.
            bitováMapa.Save(súbor.Insert(súbor.Length - 4, "I"));
        }
    }
}
```

Komentár k zdrojovému kódu: Inštancia deklarovanej triedy **Bitmapa** bude obsahovať obrazové body načítanej bitovej mapy. Cestu k požadovanej bitovej mape získa parametrický inštančný konštruktor, ktorý zabezpečuje alokáciu inštancie triedy **Bitmap** z menného priestoru **System.Drawing**. Keďže naša aplikácia je konzolová, je potrebné vložiť nielen odkaz na spomenutý menný priestor, ale tiež do projektu začleniť odkaz na zostavenie System.Drawing.dll. Programový kód verejnej inštančnej metódy **Invertovať** sa sústreďuje na inverziu bitovej mapy. Pripomeňme, že inverzia je jednou z grafických transformácií, ktorá vypočítava opačnú (inverznú) farebnú informáciu každého obrazového bodu, z ktorých je bitová mapa zložená. Inverziou obrazového bodu P_1 s farebným vektorom $[R_1, G_1, B_1]$ získame obrazový bod P_2 s farebným vektorom $[255 - R_1, 255 - G_1, 255 - B_1]$. Keď metóda **Invertovať** dokončí inverziu, uloží upravenú bitovú mapu do nového rastrového súboru (názov tohto súboru vznikne z názvu pôvodného súboru, ku ktorému metóda pridá písmeno „I“).

Deklarovanú triedu použijeme pri praktickom nasadení (uvádzame obsah zdrojového súboru Program.cs):

```
// Zdrojový kód sekvenčnej aplikácie.
using System;
using System.Collections.Generic;
```

```
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;

namespace diz
{
    class Program
    {
        static void Main(string[] args)
        {
            // Vytvorenie poľa objektov triedy Bitmapa.
            Bitmapa[] bitmapy = new Bitmapa[] {
                new Bitmapa(@"c:\Obrázky\obr_01.jpg"),
                new Bitmapa(@"c:\Obrázky\obr_02.jpg"),
                new Bitmapa(@"c:\Obrázky\obr_03.jpg"),
                new Bitmapa(@"c:\Obrázky\obr_04.jpg"),
                new Bitmapa(@"c:\Obrázky\obr_05.jpg"),
                new Bitmapa(@"c:\Obrázky\obr_06.jpg"),
                new Bitmapa(@"c:\Obrázky\obr_07.jpg"),
                new Bitmapa(@"c:\Obrázky\obr_08.jpg")};

            Console.WriteLine("Prebiehajú sekvenčné inverzie " +
               "bitových máp...");
            Stopwatch sw = new Stopwatch();
            sw.Start();
            // Sekvenčná inverzia všetkých bitových máp.
            for (int i = 0; i < bitmapy.Length; i++)
            {
                bitmapy[i].Invertovať();
            }
            sw.Stop();
            Console.WriteLine("Sekvenčné inverzie bitových máp " +
                "sú hotové [celkový čas: {0} ms].",
                sw1.Elapsed.TotalMilliseconds);
            Console.Read();
        }
    }
}
```

Komentár k zdrojovému kódu: Kód predpokladá, že na pevnom disku C existuje priečinok **Obrázky**, v ktorom sa nachádza 8 bitových máp. Program vytvorí 8 inštancií triedy **Bitmapa**, pričom každá inštancia načíta jednu bitovú mapu z priečinka. Spracovanie programu pokračuje synchrónnym vykonaním grafických transformácií všetkých bitových máp. Aplikácia meria čas, ktorý je potrebný na dokončenie inverzie bitových máp.

Sekvenčnú aplikáciu sme podrobili výkonnostným testom, a to tak v ladiacom, ako aj v ostrom režime prekladu. Pri testovaní sme použili počítač so 4-jadrovým procesorom Intel Core 2 Quad Q6600. Namerané výsledky sú zoskupené v tab. 16.1 – 16.2.

Kolekcia ladiacich testov (Debug)	**Sekvenčná verzia programu (exekučný čas v ms)**
1. výpočet	14803
2. výpočet	14778
3. výpočet	14838
Priemerný exekučný čas	**14807**

Tab. 16.1: Výkonnostné testy sekvenčnej aplikácie (ladiace zostavenie)

Kolekcia ostrých testov (Release)	**Sekvenčná verzia programu (exekučný čas v ms)**
1. výpočet	14795
2. výpočet	14793
3. výpočet	14815
Priemerný exekučný čas	**14801**

Tab. 16.2: Výkonnostné testy sekvenčnej aplikácie (ostré zostavenie)

Naším cieľom však nie je sekvenčná aplikácia, ale jej paralelný ekvivalent. Po dôkladnej analýze bázového zdrojového kódu sme konštatovali, že najjednoduchšie bude paralelizovať kód hlavnej metódy **Main**. Ďalej uvádzame zdrojový kód viacvláknovej aplikácie, ktorú sme zostrojili modifikovaním pôvodnej jednovláknovej aplikácie:

```
// Zdrojový kód paralelnej aplikácie (uložený v súbore Program.cs).
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
// Vloženie potrebného menného priestoru pre potreby paralelizácie.
using System.Threading;

namespace diz
{
    class Program
    {
        static void Main(string[] args)
        {
            Bitmapa[] bitmapy = new Bitmapa[] {
                new Bitmapa(@"c:\Obrázky\obr_01.jpg"),
                new Bitmapa(@"c:\Obrázky\obr_02.jpg"),
                new Bitmapa(@"c:\Obrázky\obr_03.jpg"),
                new Bitmapa(@"c:\Obrázky\obr_04.jpg"),
                new Bitmapa(@"c:\Obrázky\obr_05.jpg"),
                new Bitmapa(@"c:\Obrázky\obr_06.jpg"),
                new Bitmapa(@"c:\Obrázky\obr_07.jpg"),
                new Bitmapa(@"c:\Obrázky\obr_08.jpg")};

            // Vytvorenie poľa pracovných vlákien.
            Thread[] pracovnéVlákna = new Thread[] {
                new Thread(new ThreadStart(bitmapy[0].Invertovať)),
                new Thread(new ThreadStart(bitmapy[1].Invertovať)),
                new Thread(new ThreadStart(bitmapy[2].Invertovať)),
                new Thread(new ThreadStart(bitmapy[3].Invertovať)),
                new Thread(new ThreadStart(bitmapy[4].Invertovať)),
                new Thread(new ThreadStart(bitmapy[5].Invertovať)),
                new Thread(new ThreadStart(bitmapy[6].Invertovať)),
                new Thread(new ThreadStart(bitmapy[7].Invertovať)),
            };
            Stopwatch sw1 = new Stopwatch();
            Console.WriteLine("Prebiehajú paralelné inverzie " +
                "bitových máp...");
            sw1.Start();
            // Paralelná inverzia všetkých bitových máp.
            for (int i = 0; i < 8; i++)
            {
                pracovnéVlákna[i].Start();
            }
            for (int i = 0; i < 8; i++)
            {
                pracovnéVlákna[i].Join();
            }
```

```
            sw1.Stop();
            Console.WriteLine("Paralelné inverzie bitových máp " +
                "sú hotové [celkový čas: {0} ms].",
                sw1.Elapsed.TotalMilliseconds);
            Console.Read();
        }
    }
}
```

Komentár k zdrojovému kódu: Paralelizácia sekvenčnej aplikácie je vskutku intuitívna. Podstatou je explicitná tvorba poľa pracovných vlákien, pričom dbáme na to, aby sa počet pracovných vlákien zhodoval s počtom bitových máp, ktoré chceme podrobiť grafickej transformácii. Každé pracovné vlákno necháme transformovať práve jednu bitovú mapu, čím dosiahneme priaznivé rozdelenie pracovného zaťaženia. Paralelná aplikácia využíva statický prístup, kedy v čase prekladu zdrojového kódu programu prijímame rozhodnutie o celkovom počte vytvorených pracovných vlákien. Ak by sme chceli zvýšiť citlivosť aplikácie, mohli by sme uprednostniť dynamický prístup, ktorý by nám dovolil zvýšiť mieru granularity pri tvorbe pracovných vlákien aplikácie.

Testy paralelnej aplikácie na počítači so 4-jadrovým procesorom Intel Core 2 Quad Q6600 preukázali, že konverzia sekvenčnej aplikácie na paralelnú je spojená so signifikantným nárastom výkonnosti (tab. 16.3 – 16.4).

Kolekcia ladiacich testov (Debug)	Paralelná verzia programu (exekučný čas v ms)
1. výpočet	4092
2. výpočet	4045
3. výpočet	4007
Priemerný exekučný čas	**4048**

Tab. 16.3: Výkonnostné testy paralelnej aplikácie (ladiace zostavenie)

Kolekcia ostrých testov (Release)	Paralelná verzia programu (exekučný čas v ms)
1. výpočet	3959
2. výpočet	4002
3. výpočet	4173
Priemerný exekučný čas	**4045**

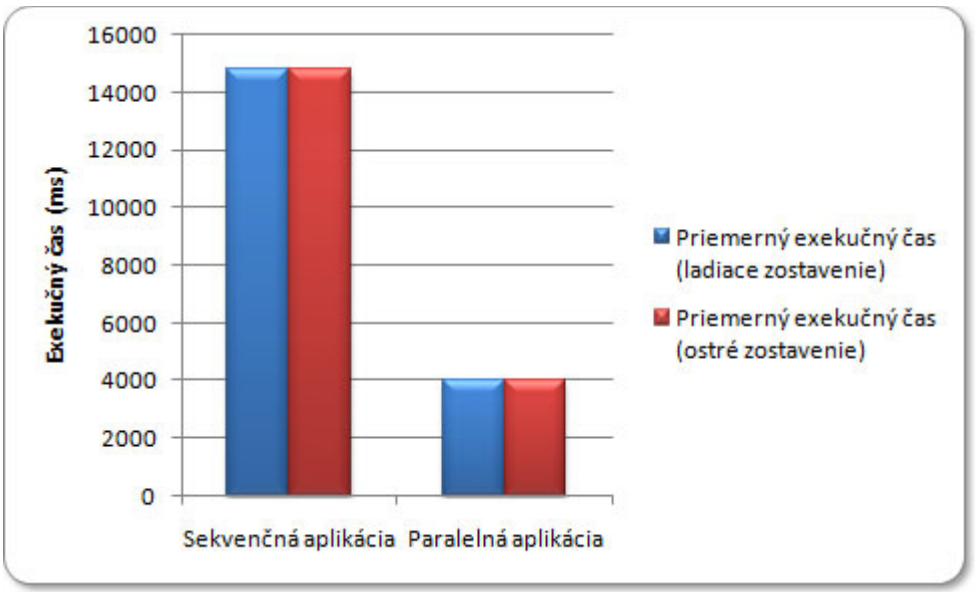
Tab. 16.4: Výkonnostné testy paralelnej aplikácie (ostré zostavenie)

Porovnanie výkonnosti sekvenčnej a paralelnej aplikácie je uvedený v tab. 16.5.

	Sekvenčná aplikácia	Paralelná aplikácia	Nárast výkonu po paralelizácii
Priemerný exekučný čas (ladiace zostavenie)	14807	4048	**3,66**
Priemerný exekučný čas (ostré zostavenie)	14801	4045	**3,66**

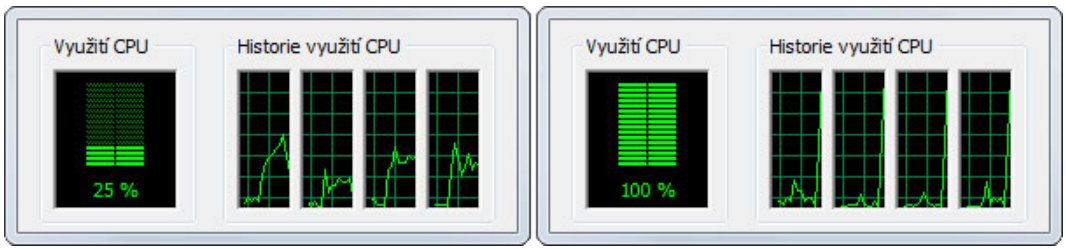
Tab. 16.5: Porovnanie výkonnosti sekvenčnej a paralelnej aplikácie

Ako je zrejmé z dosiahnutých výsledkov, pri ladiacich testoch je paralelná aplikácia 3,6-krát rýchlejšia ako sekvenčná aplikácia. Testy ostrých zostavení oboch typov aplikácií dokazujú, že paralelná aplikácia je rýchlejšia 3,6-krát. Tieto výsledky sú istotne potešujúce, najmä keď uvážime, že paralelná aplikácia manipuluje s väčším množstvom pracovných vlákien ako je počet exekučných jadier použitého viacjadrového procesora.



Obr. 16.1: Porovnanie výkonnosti sekvenčnej a paralelnej aplikácie

Keďže sekvenčná aplikácia obsahuje len jedno (primárne) programové vlákno, využíva len jedno exekučné jadro 4-jadrového procesora. Využitie výpočtových kapacít počítačového systému je štvrtinové. Naopak, paralelná aplikácia pracuje so zväzkom pracovných vlákien, ktoré budú v primeranom pomere distribuované na všetky dostupné exekučné jadrá 4-jadrového procesora. Paralelná aplikácia využije hardvérovú silu počítačovej stanice na 100 % (obr. 16.2).



Obr. 16.2: Porovnanie využitia výpočtových kapacít systému sekvenčnou (vľavo) a paralelnou aplikáciou

17 Paralelná platforma Microsoft Parallel Extensions

Paralelná platforma Microsoft Parallel Extensions[28] je nadstavbou vývojovo-exekučnej platformy Microsoft .NET Framework 3.5[29]. Platforma Parallel Extensions zavádza syntakticko-sémantické konštrukcie na vyjadrenie deklaratívneho a imperatívneho vysoko abstraktného explicitného dátového a úlohového paralelizmu. Platforma Parallel Extensions je unifikovaná pre použitie v ktoromkoľvek .NET-kompatibilnom programovacom jazyku. Unifikácia viedla k tomu, že rozhranie platformy Parallel Extensions (rozširujúce rozhranie existujúcej bázovej knižnice tried) je priamo dosiahnuteľné z jazykov C# 3.0, Visual Basic 2008, C++/CLI a ďalších, a to bez nutnosti modifikácie ich jazykových špecifikácií, resp. kompilátorov.

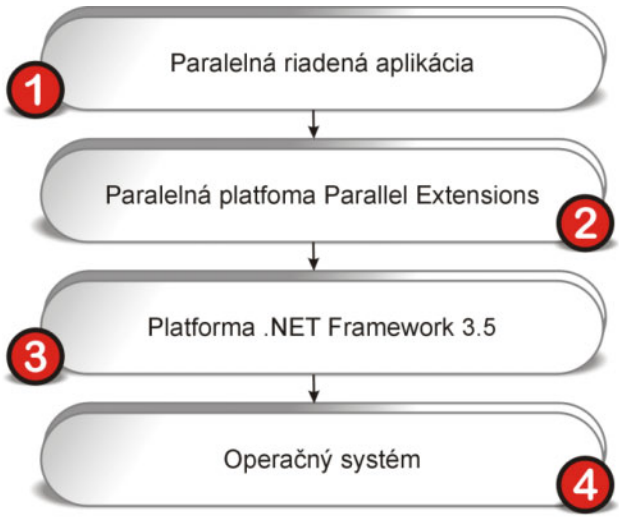
Interakčné väzby medzi paralelnou riadenou aplikáciou, paralelnou platformou Parallel Extensions, vývojovo-exekučnou platformou Microsoft .NET Framework 3.5 a operačným systémom triedy Microsoft Windows, sú znázornené na obr. 17.1.

K významným vlastnostiam paralelnej platformy Parallel Extensions patrí:

1. **Zmena programovacej paradigmy.** Na riadenú aplikáciu nemôžeme nahliadať ako na konečnú a neprázdnu množinu programových príkazov, ktoré budú vykonávané sekvenčne. Miesto toho prechádzame k ponímaniu paralelnej riadenej aplikácie ako vykonateľnej jednotky, ktorá pozostáva z množiny súbežne realizovaných úloh.

[28] V mnohých zdrojoch technických informácií sa paralelná platforma Parallel Extensions označuje svojím kódovým názvom Parallel FX.

[29] Tento jav platí pre vývojovo-exekučnú platformu Microsoft .NET Framework 3.5 a integrované vývojové prostredie Microsoft Visual Studio 2008. Počnúc vývojovo-exekučnou platformou Microsoft .NET Framework 4.0 a integrovaným vývojovým prostredím Visual Studio 2010 bude paralelná platforma Parallel Extensions úplne zakomponovaná do platformy Microsoft .NET Framework 4.0 a nebude pôsobiť ako jej nadstavba.



Obr. 17.1: Interakčné väzby riadenej aplikácie a paralelnej platformy Microsoft Parallel Extensions

2. **Vysoká úroveň abstrakcie.** Vďaka začleneniu nového aplikačného programového rozhrania smieme pracovať s vysoko abstraktnými programovými konštrukciami a entitami, ktoré podporujú paralelizáciu procesov. K abstraktným programovým konštrukciám patria napríklad cykly s podporou paralelizácie množín iterácií, paralelne vykonateľné úlohy, špeciálne vysokoúrovňové synchronizačné primitíva, alebo koordinačné dátové štruktúry.

3. **Eliminácia povinnosti explicitnej tvorby pracovných programových vlákien.** Vývoj paralelnej aplikácie sa zaobíde bez manuálnej konštrukcie pracovných programových vlákien, ako aj bez segmentácie a delegovania úloh, ktoré budú na týchto vláknach realizované. Paralelná platforma Parallel Extensions obsahuje virtuálny stroj pre paralelné spracovanie (VSPS), ktorý spolupracuje s virtuálnym plánovačom úloh (VPÚ) a virtuálnym manažérom zdrojov (VMZ).

VSPS zabezpečuje automatické vytváranie pracovných programových vlákien, pričom:

- Dbá na optimálny počet pracovných vlákien.
- Zabezpečuje optimálnu distribúciu pracovného zaťaženia medzi jednotlivými pracovnými vláknami.
- Garantuje automatické riadenie životných cyklov pracovných vlákien.

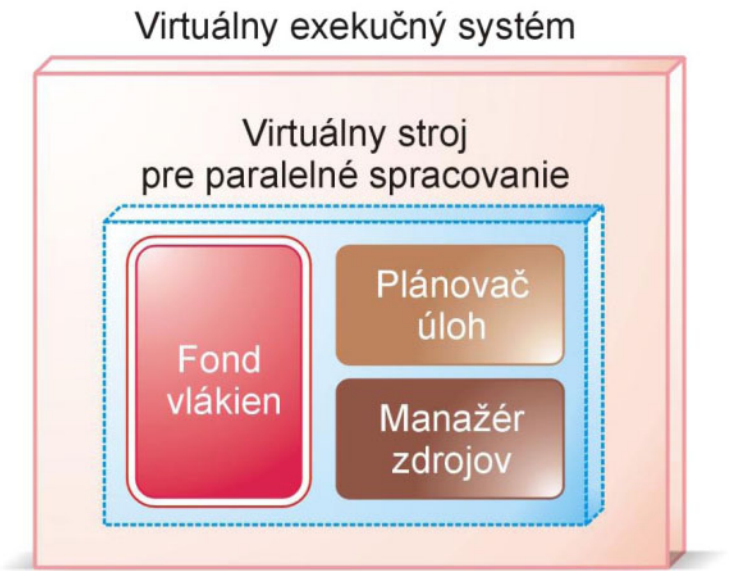
VSPS interne spravuje fond programových vlákien a podľa potreby recykluje vlákna, čím maximalizuje mieru ich opätovného využitia a naopak, minimalizuje alokáciu systémových prostriedkov.

VSPS pracuje v rámci virtuálneho exekučného systému vývojovo-exekučnej platformy Microsoft .NET Framework 3.5 (obr. 17.2).

4. **Efektívnosť, robustnosť a škálovateľnosť paralelných aplikácií.** VSPS nachádza kontextovo závislú, no pritom maximálne efektívnu väzbu medzi počtom programových vlákien viacvláknovej riadenej aplikácie a celkovou výpočtovou kapacitou počítačového systému (čiže počtom exekučných jadier pri počítači s viacjadrovým procesorom, resp. počtom procesorov pri viacprocesorovom stroji).

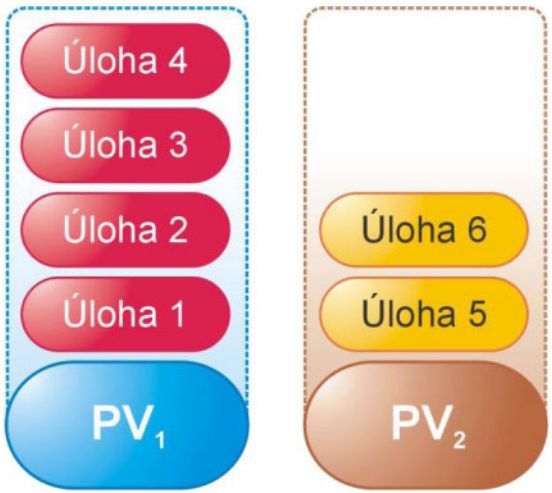
V záujme čo možno najefektívnejšej exekúcie riadenej paralelnej aplikácie využíva VSPS v spolupráci s VPÚ mnohé sofistikované techniky, napríklad transport úloh naprieč jednotlivými pracovnými vláknami[30]. Transport úloh sa využíva preto, aby boli všetky alokované pracovné programové vlákna využité. Cieľom je eliminovať čas nečinnosti pracovných vlákien. Techniku transportu úloh medzi vláknami budeme demonštrovať na nasledujúcom príklade.

[30] V origináli sa technika, ktorú my nazývame „transport úloh“, volá „work stealing“, doslova „odcudzenie práce“.



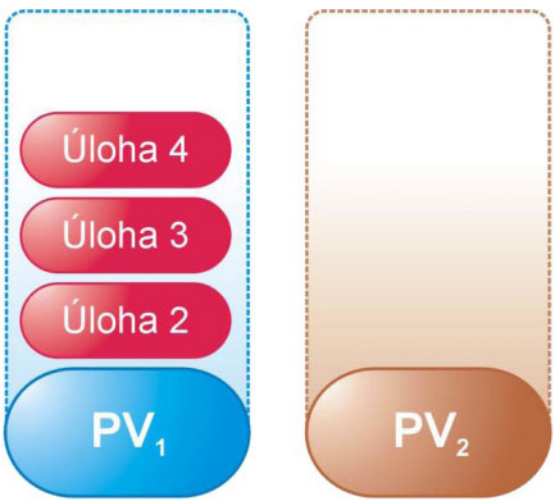
Obr. 17.2: Vzťah medzi virtuálnym strojom pre paralelné spracovanie (VSPS) a virtuálnym exekučným systémom (VES)

Príklad: Predpokladajme, že VSPS podľa špecifikácie 4-prvkovej množiny paralelne realizovaných úloh (manažovaných pomocou VPÚ) skonštruuje 2 pracovné programové vlákna. VSPS rozhodne, že na 1. pracovné vlákno ($\mathbf{PV_1}$) umiestni 4 úlohy, kým 2. pracovné vlákno ($\mathbf{PV_2}$) zaťaží spracovaním 2 úloh (obr. 17.3).



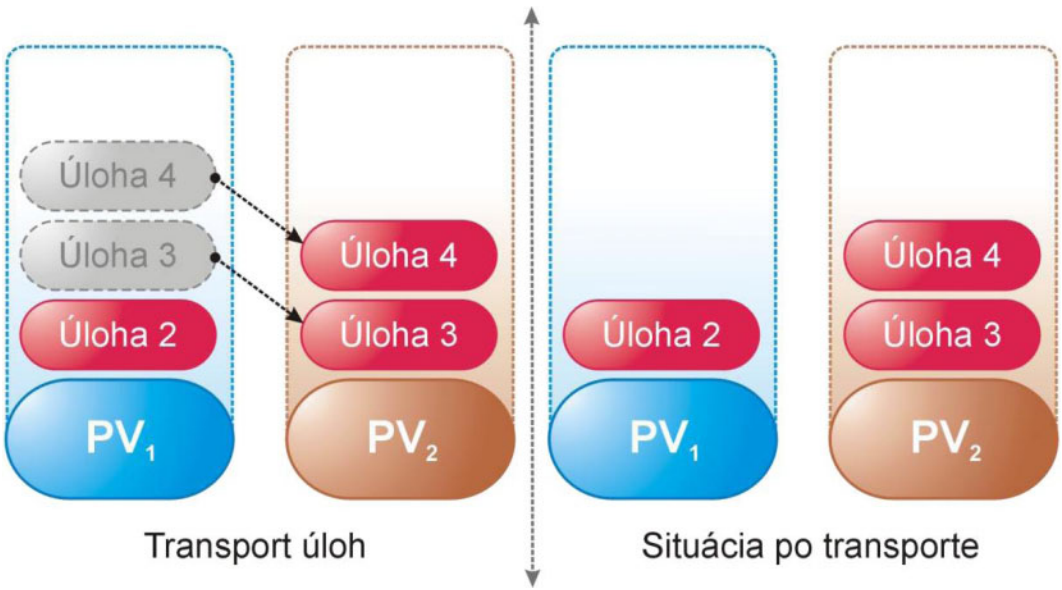
Obr. 17.3: Transport úloh medzi vláknami (1. štádium)

Predpokladajme, že zložitosť úloh, ktoré vykonáva 2. pracovné vlákno je nižšia ako zložitosť úloh, ktoré realizuje 1. pracovné vlákno. Za týchto podmienok bude v určitom čase (t_n) obraz pracovných vlákien takýto: 1. pracovné vlákno vykonalo 1 zo 4 delegovaných úloh a 2. pracovné vlákno vykonalo všetky úlohy, ktoré malo uskutočniť (obr. 17.4).



Obr. 17.4: Transport úloh medzi vláknami (2. štádium)

Keď je 2. pracovné vlákno hotové so svojimi úlohami, ostáva nečinné, pretože neexistujú žiadne ďalšie úlohy, ktorých realizáciu by malo 2. pracovné vlákno v kompetencii. Aby VSPS zabránil zotrvaniu 2. pracovného vlákna v neaktívnom stave, uskutoční transport 2 úloh z 1. pracovné vlákna na 2. pracovné vlákno. Po transporte bude 1. pracovné vlákno zaneprázdnené spracovaním 1 úlohy, zatiaľ čo druhé vlákno bude uskutočňovať 2 zverené úlohy (obr. 17.5).



Obr. 17.5 Transport úloh medzi vláknami (3. štádium)

Technika transportu úloh medzi viacerými pracovnými vláknami má tieto silné stránky:

1. Zabezpečuje približne rovnomernú distribúciu pracovného zaťaženia medzi pracovnými vláknami.
2. Minimalizuje stavy nečinnosti pracovných vlákien.
3. Pri transporte úloh sa venuje zvláštna pozornosť lokalizácii dátových objektov, pričom kvôli efektívnej práci s pamäťou sú uprednostňované úlohy, ktorých dátové objekty sú v niektorej z rýchlych vyrovnávacích pamätí (L_1-cache, L_2-cache alebo L_3-cache).

Paralelná platforma Parallel Extensions ponúka vývojárom viacero variantov vysokoúrovňového explicitného paralelizmu:

1. **Deklaratívny dátový paralelizmus.** Deklaratívny dátový paralelizmus je uplatňovaný pri spracúvaní dopytov, ktoré sú paralelne odosielané viacerým entitám (objektom, inicializovaným poliam a kolekciám). Komponent paralelnej platformy Parallel Extensions, ktorý zabezpečuje deklaratívny prístup k práci s dátami, sa volá Parallel Language Integrated Query (rovnako sa používajú aj skrátené názvy Parallel LINQ a PLINQ).

2. **Imperatívny dátový paralelizmus.** Imperatívny dátový paralelizmus je aplikovaný pri paralelizácii iteratívnych príkazov s automatickou distribúciou paralelne vykonávaných iteračných množín. Programové konštrukcie na podporu imperatívneho dátového paralelizmu sú združené v knižnici Task Parallel Library (TPL).

3. **Imperatívny úlohový paralelizmus.** Imperatívny úlohový paralelizmus sa využíva pri návrhu a implementácii úloh, resp. programových činností, ktoré majú byť vykonávané súbežne. Programové konštrukcie na podporu imperatívneho úlohového paralelizmu sú združené v knižnici TPL.

4. **Paralelizmus dátových tokov.** Ak je výstup jednej programovej operácie vstupom pre inú programovú operáciu, môžeme implementovať reťazec aktérov, ktorí budú na báze modelu producent/konzument uskutočňovať paralelizmus dátových tokov. Programové konštrukcie na podporu paralelizmu dátových tokov sa nachádzajú v knižnici TPL.

17.1 Microsoft Parallel Extensions: Praktické cvičenie

Pri praktickom testovaní paralelnej platformy Microsoft Parallel Extensions sme sa venovali paralelizácii pôvodne sekvenčnej aplikácie, ktorá vyhľadáva všetky prvočísla z intervalu $<2, 10^7>$.

Zdrojový kód sekvenčnej aplikácie vyhľadávajúcej prvočísla vyzeral takto:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;

namespace diz
{
    class Program
    {
        static void Main(string[] args)
        {
            int hornáHranica = 10000000, početPrvočísel = 0;
            Stopwatch sw = new Stopwatch();
            Console.WriteLine("Vyhľadávam sekvenčne prvočísla...");
            sw.Start();

            // Sekvenčné vyhľadávanie prvočísel.
            for (int i = 2; i <= hornáHranica; i++)
            {
                if (Test(i)) početPrvočísel++;
            }

            Console.WriteLine("Vyhľadávanie sa skončilo.");
            sw.Stop();
            Console.WriteLine("V intervale <2, {0}> je {1} prvočísel.",
                hornáHranica, početPrvočísel);
            Console.WriteLine("Exekučný čas (ms): {0}.",
                sw.ElapsedMilliseconds);
            Console.Read();
        }
        // Metóda, ktorá uskutočňuje test prvočíselnosti.
        static bool Test(int číslo)
        {
            int odmocninaZČísla = (int)Math.Sqrt(číslo);
            for (int i = 2; i <= odmocninaZČísla; i++)
            {
                if (číslo % i == 0) return false;
            }
            return true;
        }
    }
}
```

Komentár k zdrojovému kódu: Sekvenčná aplikácia zisťuje počet prvočísel na základe testu prvočíselnosti, ktorý realizuje nasledujúci matematický algoritmus:

1. Nech n je prirodzené vstupné číslo.

2. Skontrolujeme, či je číslo n deliteľné ktorýmkoľvek prirodzeným číslom m z intervalu $< 2, \sqrt{n} >$.

3. Ak je n deliteľné ľubovoľným m, potom n je zložené číslo, inak je n prvočíslo.

Algoritmus diagnostikuje, či je testované číslo prvočíslo, alebo zložené číslo. Ak metódu, ktorá implementuje spomenutý algoritmus, aktivujeme na všetkých číslach z požadovaného číselného intervalu, získame celkový počet prvočísel, ktoré sa v danom intervale nachádzajú.

Sekvenčná aplikácia vyhľadáva prvočísla pomocou cyklu **for**, pričom v každej iterácii cyklu volá metódu **Test**. Táto parametrická metóda prevezme odovzdané číslo a preverí, či ide o prvočíslo, alebo zložené číslo. V prípade, ak je analyzované číslo prvočíslom, metóda **Test** vracia logickú pravdu, inak je návratovou hodnotou metódy logická nepravda.

Výkonnosť sekvenčnej aplikácie sme testovali v troch samostatných reláciách na počítači so 4-jadrovým procesorom Intel Core 2 Quad. Namerané výsledky, ktoré sekvenčná aplikácia dosahovala v ladiacich aj ostrých zostaveniach, uvádzame v tab. 17.1 – 17.2.

Kolekcia ladiacich testov (Debug)	Sekvenčná verzia programu (exekučný čas v ms)
1. relácia	16848
2. relácia	16720
3. relácia	16748
Priemerný exekučný čas	**16772**

Tab. 17.1: Výkonnostné testy sekvenčnej aplikácie (ladiace zostavenie)

Kolekcia ostrých testov (Release)	Sekvenčná verzia programu (exekučný čas v ms)
1. relácia	16332
2. relácia	16328
3. relácia	16400
Priemerný exekučný čas	**16354**

Tab. 17.2: Výkonnostné testy sekvenčnej aplikácie (ostré zostavenie)

Testy dokázali, že sekvenčná aplikácia potrebuje na nájdenie radu prvočísel viac ako 16 sekúnd exekučného času.

Sekvenčnú aplikáciu sme preto upravili tak, aby využívala možnosti paralelnej platformy Microsoft Parallel Extensions. Po modifikácii sme získali paralelnú aplikáciu, ktorej zdrojový kód vyzerá takto:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;
using System.Threading;
```

```
namespace diz
{
    class Program
    {
        static void Main(string[] args)
        {
            int hornáHranica = 10000000, pocetPrvočísel = 0;
            Stopwatch sw = new Stopwatch();
            Console.WriteLine("Vyhľadávam paralelne prvočísla...");
            sw.Start();

            // Paralelné vyhľadávanie prvočísel.
            Parallel.For(2, hornáHranica + 1, i =>
            {
                if (Test(i)) Interlocked.Increment(ref pocetPrvočísel);
            });

            Console.WriteLine("Vyhľadávanie sa skončilo.");
            sw.Stop();
            Console.WriteLine("V intervale <2, {0}> je {1} prvočísel.",
                hornáHranica, pocetPrvočísel);
            Console.WriteLine("Exekučný čas (ms): {0}.",
                sw.ElapsedMilliseconds);
            Console.Read();
        }
        static bool Test(int číslo)
        {
            int odmocninaZČísla = (int)Math.Sqrt(číslo);
            for (int i = 2; i <= odmocninaZČísla; i++)
            {
                if (číslo % i == 0) return false;
            }
            return true;
        }
    }
}
```

Komentár k zdrojovému kódu: Paralelizácia aplikácie si vyžadovala zmeniť sekvenčný cyklus **for** za paralelný cyklus **for**. Z technického hľadiska to znamená, že cyklus **for** sme nahradili volaním statickej metódy **For** triedy **Parallel** z menného priestoru **System.Threading**[31]. Statickej metóde **For** odovzdávame hranice číselného intervalu a λ-výraz, ktorý aktivuje metódu vykonávajúcu test

[31] Pripomíname, že aby zdrojový kód spoľahlivo fungoval, musí aplikačný projekt obsahovať odkaz na zostavenie System.Threading.dll paralelnej platformy Microsoft Parallel Extensions.

prvočíselnosti. Hoci v budúcich verziách paralelnej platformy Microsoft Parallel Extensions už bude situácia zrejme iná, inkrementáciu premennej **početPrvočísel** musíme explicitne naprogramovať ako atomickú operáciu[32].

Ako si paralelná aplikácia viedla v praktických testoch na počítači so 4-jadrovým procesorom Intel Core 2 Quad, dokumentujú tab. 17.3 – 17.4.

Kolekcia ladiacich testov (Debug)	**Paralelná verzia programu (exekučný čas v ms)**
1. relácia	4532
2. relácia	4778
3. relácia	4701
Priemerný exekučný čas	**4671**

Tab. 17.3: Výkonnostné testy paralelnej aplikácie (ladiace zostavenie)

Kolekcia ostrých testov (Release)	**Paralelná verzia programu (exekučný čas v ms)**
1. relácia	4476
2. relácia	4654
3. relácia	4549
Priemerný exekučný čas	**4560**

Tab. 17.4: Výkonnostné testy paralelnej aplikácie (ostré zostavenie)

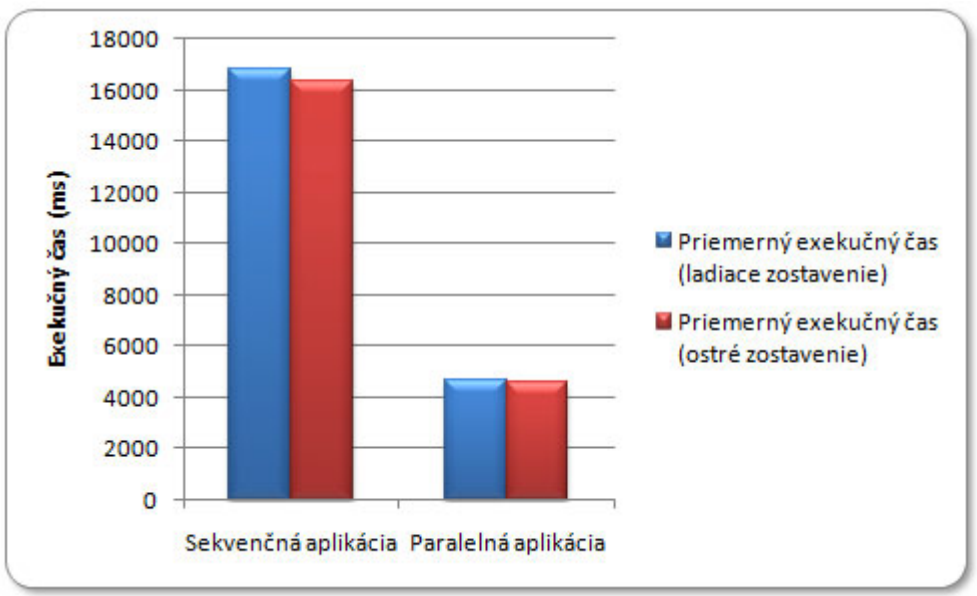
[32] Paralelný cyklus **for** sa v CTP verzii paralelnej platformy Microsoft Parallel Extensions z júna 2008 nespráva korektne pri prístupe k zdieľaným prostriedkom (v našom prípade je týmto prostriedkom lokálna premenná, na ktorú však nie je aplikovaná redukcia). Aby sme predišli nedeterministickému výstupu, atomicky inkrementujeme lokálnu premennú **početPrvočísel**.

Komparáciu výkonnosti sekvenčnej a paralelnej aplikácie prináša tab. 17.5.

	Sekvenčná aplikácia	**Paralelná aplikácia**	**Nárast výkonu po paralelizácii**
Priemerný exekučný čas (ladiace zostavenie)	16772	4671	**3,59**
Priemerný exekučný čas (ostré zostavenie)	16354	4560	**3,59**

Tab. 17.5: Porovnanie výkonnosti sekvenčnej a paralelnej aplikácie

Paralelizácia sekvenčnej aplikácie zvýšila jej výkonnosť približne 3,6-krát. Výstup paralelnej aplikácie môžeme očakávať za asi 4,5 sekundy, zatiaľ čo na odpoveď sekvenčnej aplikácie sme boli nútení čakať viac ako 16 sekúnd. Po vykonaní testov môžeme konštatovať, že aplikácia zaznamenáva po svojej paralelizácii sublineárny nárast výkonnosti.



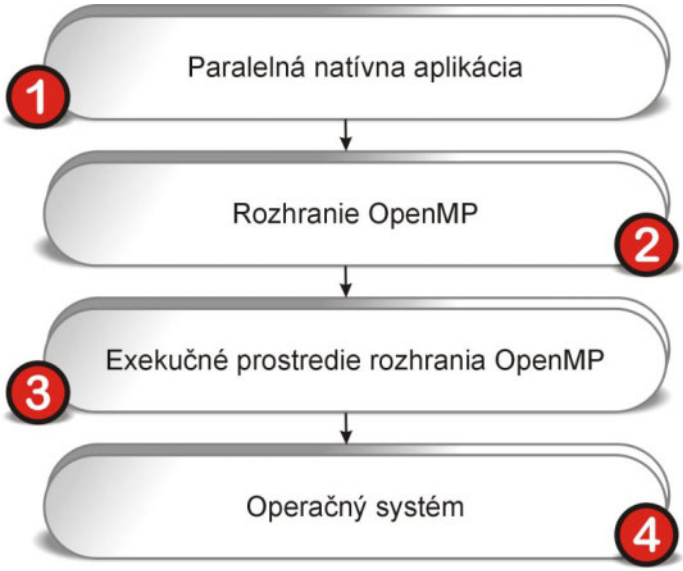
Obr. 17.6: Porovnanie výkonnosti sekvenčnej a paralelnej aplikácie

18 OpenMP – natívne rozhranie na podporu paralelizácie výpočtových procesov

OpenMP[33] je multiplatformové aplikačné programové rozhranie (API), poskytujúce vývojárom nástroje na implementáciu explicitného paralelizmu so strednou úrovňou abstrakcie. Rozhranie OpenMP zavádza súpravu direktív pre kompilátor, ktorými môžu programátori priamo ohraničiť tie programové bloky softvérových aplikácií, ktoré majú byť realizované paralelne. Okrem toho zahŕňa rozhranie OpenMP vlastnú knižnicu, sústavu premenných prostredia a exekučné prostredie na podporu realizácie paralelných výpočtov. Vzťah medzi softvérovou aplikáciou, rozhraním OpenMP, spriazneným exekučným prostredím a operačným systémom je uvedený na obr. 18.1.

Rozhranie OpenMP je určené predovšetkým pre natívne programovacie jazyky, ku ktorým patrí C, C++ a Fortran. Štandard rozhrania OpenMP bol po prvýkrát formulovaný v roku 1997 (ako verzia 1.0) pre jazyk Fortran, pričom o rok neskôr bol spracovaný aj štandard pre jazyky C a C++. V roku 2000 vznikla verzia 2.0 rozhrania OpenMP pre jazyk Fortran nasledovaná edíciou pre jazyky C/C++ v roku 2002. V roku 2005 bola uvedená verzia 2.5 už pre všetky tri hlavné programovacie jazyky. Najnovšie vydanie štandardu rozhrania OpenMP sa uskutočnilo v roku 2008 a táto verzia rozhrania nesie označenie OpenMP 3.0.

[33] Názov rozhrania OpenMP je skratkou viacslovného spojenia „Open Multi-Processing“.



Obr. 18.1: Korelácia medzi softvérovou aplikáciou,
rozhraním OpenMP a operačným systémom

Rekognícia direktív, s ktorými rozhranie OpenMP spolupracuje, bola začlenená do mnohých populárnych kompilátorov jazykov C, C++ a Fortran. Hoci sa v tejto publikácii venujeme vývojovému prostrediu Microsoft Visual Studio 2008, je nutné podotknúť, že prekladače s podporou rozhrania OpenMP produkujú aj ďalšie softvérové spoločnosti, napríklad Intel, Sun či Borland/CodeGear/Embarcadero.

Práca s rozhraním OpenMP vyzerá z pohľadu tvorcu počítačovej aplikácie takto:

1. Najskôr sa použitím sofistikovaných diagnostických a monitorovacích nástrojov uskutoční dôkladná analýza životného cyklu aplikácie. Táto etapa je veľmi dôležitá najmä pri vývoji stredne veľkých a veľkých softvérových aplikácií. Automatizáciu diagnosticko-monitorovacích činností spoľahlivo vykonajú nástroje dynamickej analýzy, ako napríklad Intel VTune Performance Analyzer či AMD CodeAnalyst.

2. Potom sa ohraničia časti aplikácie s vysokou frekvenciou aktívneho využitia. Tieto časti sú predmetom optimalizačného procesu, v ktorom je nutné identifikovať bloky zdrojového kódu, ktorých efektívnosť spracovania môže byť zvýšená prostredníctvom paralelizácie.

3. Každý blok, ktorý chceme spracúvať paralelne, označíme špeciálnymi direktívami rozhrania OpenMP. Napríklad, na paralelizáciu iteratívnych príkazov **for** smieme aplikovať direktívu **#pragma omp parallel for**, voliteľne aj s komplementárnymi klauzulami a príznakmi. Blok determinovaný direktívou **#pragma** je zo strany rozhrania OpenMP považovaný za tzv. štruktúrovaný blok s paralelnou konštrukciou (v praxi sa často takýto blok označuje skrátene termínom „paralelná konštrukcia“ alebo „paralelná sekcia“). Kompilátor automaticky generuje inštrukcie, ktoré uskutočňujú nasledujúce akcie:

   - **Vytvárajú optimálny počet pracovných programových vlákien.** Pracovné vlákna sú získavané z fondu pracovných vlákien. Fond pracovných vlákien má natívny charakter a jeho manažment má na starosti exekučné prostredie rozhrania OpenMP.

   - **Samočinne delegujú úlohy na príslušné pracovné programové vlákna, pričom prihliadajú na optimálnu distribúciu pracovného zaťaženia medzi týmito vláknami.** Ak chceme napríklad paralelne vykonávať množinu príkazov umiestnených v tele cyklu, exekučné prostredie rozhrania OpenMP samočinne rozhodne o tom, koľko zväzkov paralelne vykonávaných iterácií cyklu bude vytvorených a ako budú tieto zväzky mapované na alokované pracovné vlákna. Uveďme praktický príklad: Nech je daný cyklus s 2000 iteráciami a nech je splnená podmienka, že jednotlivé iterácie cyklu sú od seba nezávislé[34]. V závislosti od výpočtovej

[34] Podmienka nezávislosti iterácií cyklu je tzv. nutnou podmienkou, ktorá musí byť splnená vždy, ak chceme implementovať techniku bezpečného paralelného spracovania množiny príkazov umiestnených v tele cyklu.

kapacity počítačového systému, na ktorom je naša aplikácia spustená, sa exekučné prostredie rozhrania OpenMP môže rozhodnúť pre tvorbu *n* paralelne vykonávaných zväzkov iterácií cyklu. V záujme konkrétnosti predpokladajme, že aplikácia beží na počítači so 4-jadrovým procesorom. Potom OpenMP skonštruuje 4 zväzky iterácií, ktoré budú súbežne vykonávané na 4 pracovných vláknach pridelených z fondu. Pre dosiahnutie optimálneho pracovného zaťaženia je nutné, aby každý zväzok obsahoval 500 iterácií cyklu.

- **Riadia životné cykly pracovných programových vlákien.** Rozhranie OpenMP využíva služby svojho exekučného prostredia, ktoré zaobstaráva kompletnú správu životných cyklov pracovných vlákien. Podstatu exekučného prostredia rozhrania OpenMP a jeho úlohu v procese riadenia životných cyklov aplikácií podrobnejšie rozoberáme v kapitole *18.1 Exekučné prostredie rozhrania OpenMP a exekúcia paralelných aplikácií*.

4. Na tomto mieste považujeme za dôležité uviesť, že v mnohých prípadoch práca vývojára spočíva iba v identifikácii paralelne vykonávaných blokov zdrojového kódu. Ak je kompilátor kompatibilný so štandardom OpenMP, pri preklade deteguje všetky paralelné konštrukcie a automaticky generuje zodpovedajúce nízkoúrovňové programové inštrukcie s podporou realizácie paralelných výpočtov. V prípade, ak kompilátor nepodporuje štandard OpenMP, nič sa nedeje, pretože direktívy sú ignorované. Za týchto okolností však bude výsledným produktom rýdzo sekvenčná aplikácia.

18.1 Exekučné prostredie rozhrania OpenMP a exekúcia paralelných aplikácií

Exekučné prostredie rozhrania OpenMP je zodpovedné za korektnú exekúciu paralelných konštrukcií počítačovej aplikácie. Proces riadenia životných cyklov sekvenčno-paralelných aplikácií sa skladá z nasledujúcich etáp:

1. Aplikácia začína svoj život ako jednovláknová. Programové príkazy sú na primárnom vlákne vykonávané sekvenčne až do okamihu, kedy bude detegovaná prvá paralelná konštrukcia. V tejto chvíli vysiela OpenMP smerom na exekučné prostredie požiadavku na pridelenie určitého množstva pracovných vlákien z natívneho fondu vlákien.

2. Po alokácii pracovných vlákien sa mení stav pôvodne jednovláknovej aplikácie na aplikáciu viacvláknovú. V ďalšom štádiu bude do každého pracovného vlákna injektovaných toľko inštrukcií, aby bola garantovaná optimálna distribúcia pracovného zaťaženia medzi vláknami.

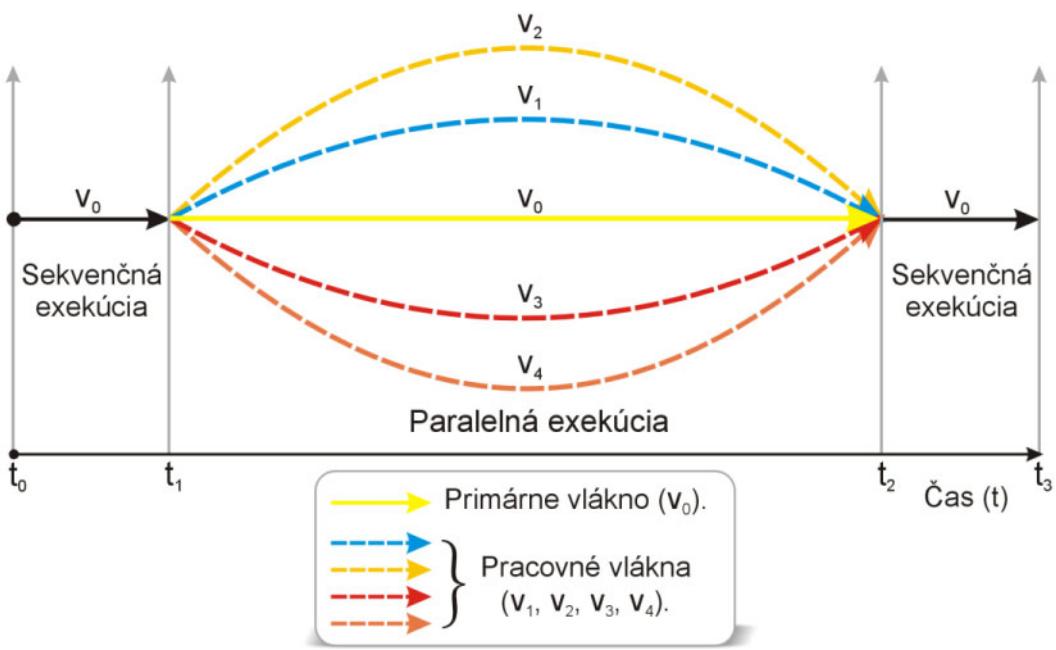
3. Exekučné prostredie rozhrania OpenMP v kooperácii s operačným systémom vytvorí väzbu medzi pracovnými vláknami aplikácie, vláknami jadra operačného systému a hardvérovými vláknami exekučných jadier viacjadrového procesora. Implicitne sa exekučné prostredie rozhrania OpenMP snaží vytvárať toľko pracovných vlákien, aby medzi nimi a exekučnými jadrami procesora existovala relácia typu 1:1. Exekúcia viacvláknovej aplikácie je v tomto momente paralelná.

4. Paralelná exekúcia pokračuje až dovtedy, pokiaľ nie sú vykonané všetky úlohy, ktoré boli delegované na pracovné vlákna. Exekučné prostredie rozhrania OpenMP vie, kde sa nachádza koniec paralelnej konštrukcie. Ak niektoré z pracovných vlákien dosiahne tento koniec skôr ako iné, toto vlákno čaká, kým koniec paralelnej konštrukcie nedosiahnu všetky ostatné pracovné vlákna. Koniec paralelnej konštrukcie kreuje implicitnú bariéru, na ktorej sa paralelná exekúcia paralelnej konštrukcie končí. Keď sú všetky

pracovné vlákna hotové so svojou prácou, pričom dosiahli bariéru, exekúcia našej aplikácie sa mení z paralelnej späť na sekvenčnú (resp. z viacvláknovej na jednovláknovú).

5. Dĺžka trvania sekvenčnej exekúcie aplikácie je determinovaná vzdialenosťou ďalšej paralelnej konštrukcie, pri detekcii ktorej zažije aplikácia ďalší tranzit do stavu paralelnej exekúcie. Pracovné vlákna získavané z natívneho fondu vlákien sú recyklované. Tieto vlákna sú vytvárané len raz (pri spracúvaní prvej paralelnej sekcie) a v ďalších etapách životného cyklu aplikácie sú opätovne využívané.

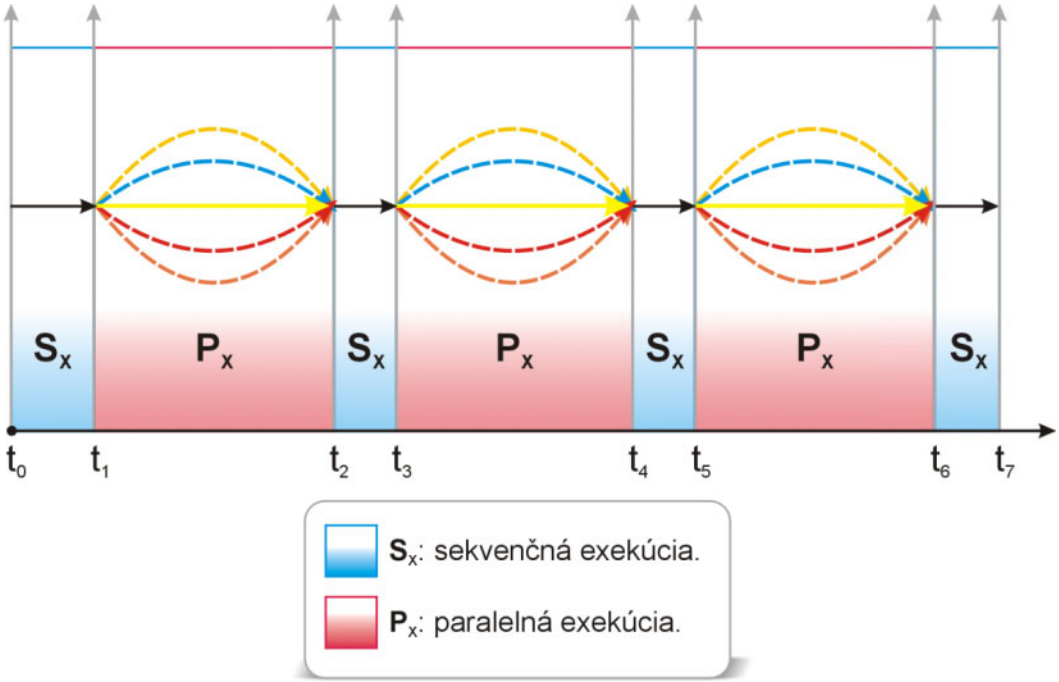
6. Ak dôjde k ukončeniu života aplikácie, automaticky sú zlikvidované všetky pracovné vlákna a exekučné prostredie rozhrania OpenMP je uvoľnené z operačnej pamäte.



Obr. 18.2: Exekúcia paralelnej konštrukcie natívnej aplikácie s podporou exekučného prostredia rozhrania OpenMP

Komentár k obr. 18.2: Softvérová aplikácia zahajuje svoj život v čase t_0. Až do času t_1 pôsobí aplikácia ako jednovláknová s jedným primárnym vláknom ($\mathbf{v_0}$) a jej exekučný model je sekvenčný. V čase t_1 aplikácia začína spracúvať prvú paralelnú konštrukciu. Spracovanie paralelnej konštrukcie mení exekučný model aplikácie zo sekvenčného na paralelný. Exekučné prostredie rozhrania OpenMP vytvára v čase t_1 4-prvkovú množinu pracovných vlákien ($\mathbf{v_1}$, $\mathbf{v_2}$, $\mathbf{v_3}$ a $\mathbf{v_4}$). Tieto pracovné vlákna vykonávajú súbežne činnosti, ktoré im boli delegované. Paralelná exekúcia aplikácie sa končí v čase t_2, teda v bode implicitnej bariéry paralelnej sekcie. Vo chvíli, keď všetky z pracovných vlákien dosiahnu bariéru paralelnej sekcie, exekučný model aplikácie sa mení z paralelného na sekvenčný. Sekvenčné spracovanie aplikácie pokračuje až do času t_3.

Absolútna početnosť výskytu paralelných konštrukcií v zdrojovom kóde aplikácie v priamej miere ovplyvňuje počet prechodov medzi jej sekvenčným a paralelným exekučným modelom. Napríklad, na obr. 18.3 je schematicky zobrazený životný cyklus aplikácie s tromi paralelnými konštrukciami.



Obr. 18.3: Exekúcia viacerých paralelných konštrukcií natívnej aplikácie s podporou exekučného prostredia rozhrania OpenMP

Komentár k obr. 18.3: V analyzovanom životnom cykle softvérovej aplikácie sa vyskytujú tieto etapy:

1. Etapy sekvenčnej exekúcie (**S_X**): sekvenčne je aplikácia spracúvaná v týchto časových intervaloch: $<t_0, t_1)$, $<t_2, t_3)$, $<t_4, t_5)$ a $<t_6, t_7)$.

2. Etapy paralelnej exekúcie (**P_X**): paralelne je aplikácia spracúvaná v týchto časových intervaloch: $<t_1, t_2)$, $<t_3, t_4)$ a $<t_5, t_6)$.

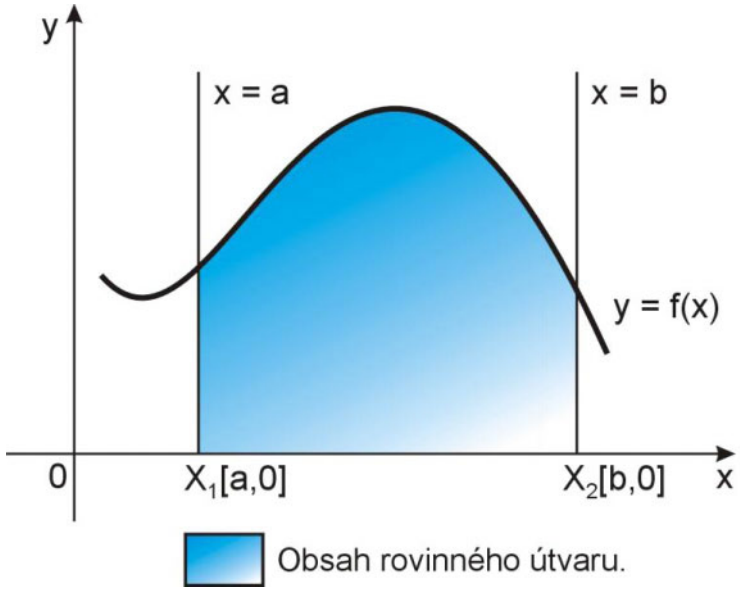
18.2 Praktické použitie rozhrania OpenMP pri implementácii paralelného algoritmu numerickej integrácie v jazyku C++

Praktickú aplikáciu použitia rozhrania OpenMP si predvedieme na implementácii algoritmu, ktorý bude realizovať numerickú integráciu. Numerická analýza definuje numerickú integráciu ako aproximačnú metódu výpočtu určitého integrálu $\int_a^b f(x)dx$. Pritom predpokladáme, že funkcia $y = f(x)$ je na intervale $< a, b >$ spojitá a nezáporná.

Hodnota určitého intervalu $\int_a^b f(x)dx$ predstavuje obsah (S) rovinného útvaru (T), ktorý je:

- zhora ohraničený grafom funkcie $y = f(x)$,
- zdola ohraničený osou x,
- zľava ohraničený priamkou rovnobežnou s osou y a prechádzajúcou bodom $X_1[a, 0]$,
- sprava ohraničený priamkou rovnobežnou s osou y a prechádzajúcou bodom $X_2[b, 0]$.

Na obr. 18.4 je znázornený rovinný útvar, ktorého obsah determinuje určitý interval $\int_{X_1}^{X_2} f(x)dx$.



Obr. 18.4: Geometrický význam určitého integrálu

Metód numerickej integrácie je viacero. My sme zvolili obdĺžnikovú metódu, pri ktorej postupujeme nasledujúcim spôsobom:

1. Interval $< X_1, X_2 >$ rozdelíme pomocou deliacich bodov $X_1 = c_0, c_1, \ldots, c_n = X_2$ na konečný počet parciálnych intervalov, pričom platí, že $c_0 < c_1 < \ldots < c_n$.

2. V každom intervale $< c_{i-1}, c_i >$ vypočítame hodnotu p funkcie $f(x)$, pričom $p = f(\frac{c_{i-1}+c_i}{2})$.

3. Pre každý interval $< c_{i-1}, c_i >$ vypočítame obsah obdĺžnika so stranami $a = c_i - c_{i-1}$ a $b = p$. Obsah i-teho obdĺžnika (S_i) určíme takto:

$$S_i = a \times b = (c_i - c_{i-1}) \times f(\frac{c_{i-1} + c_i}{2})$$

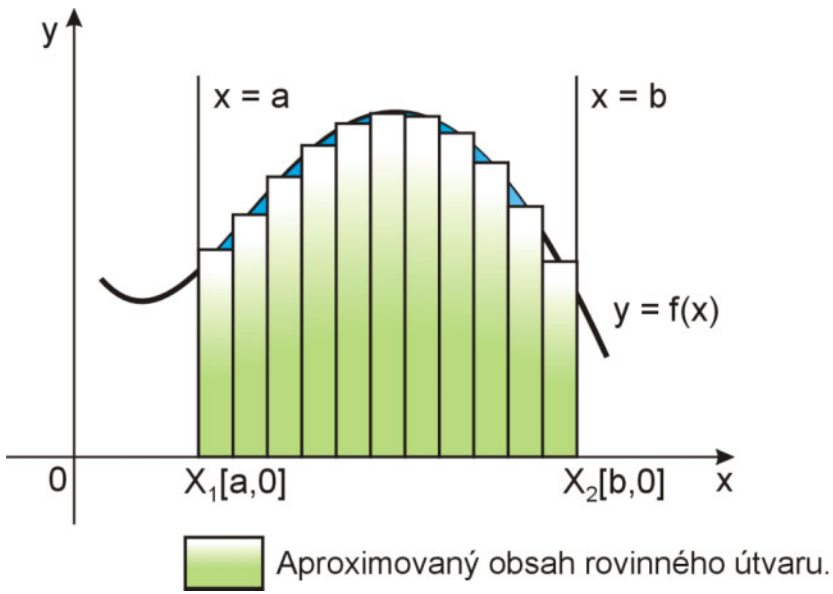
4. Keď spočítame obsahy obdĺžnikov pre všetky parciálne intervaly, získame aproximovanú hodnotu určitého integrálu. Zapísané matematicky:

$$S = \sum_{i=1}^{n} (c_i - c_{i-1}) \times f(\frac{c_{i-1} + c_i}{2})$$

5. Použitím obdĺžnikovej metódy nakoniec dospejeme k nasledujúcej približnej rovnosti:

$$\int_{X_1}^{X_2} f(x)dx \cong S$$

Vizualizáciu aproximovaného výpočtu hodnoty určitého intervalu $\int_{X_1}^{X_2} f(x)dx$ uvádzame na obr. 18.5.



Obr. 18.5: Numerická integrácia pomocou obdĺžnikovej metódy

V našej praktickej aplikácii použijeme numerickú integráciu na výpočet aproximovanej hodnoty určitého integrálu $\int_0^1 \frac{4}{1+x^2} dx$. Ak použijeme dostatočný počet parciálnych intervalov, vypočítaná hodnota určitého integrálu bude približne rovná Ludolfovmu číslu (π), teda $\int_0^1 \frac{4}{1+x^2} dx \cong \pi$.

Najskôr uvedieme sekvenčnú verziu algoritmizácie predmetnej numerickej integrácie v jazyku C++:

```
#include <iostream>
#include <windows.h>

using namespace std;

int main()
{
    double krok, x, obsah = 0.0f;
    long int i, pocetKrokov = 1000000000;
    unsigned int cas0, cas1;
    krok = 1.0f / pocetKrokov;

    for(int n = 1; n <= 3; n++)
    {
        x = obsah = 0.0f;
        cas0 = GetTickCount();
        cout << "Vypocet c. " << n << " bol zahajeny." << endl;
        for(i = 0; i < pocetKrokov; i++)
        {
            x = i * krok;
           obsah += 4.0f / (1.0f + x * x);
        }
        cas1 = GetTickCount();
        cout << "Vypocet c. " << n << " bol skonceny." << endl;
        cout.precision(20);
        cout << "Vystup: " << obsah * krok << endl;
        cout << "Cas (ms): " << cas1 - cas0 << endl << endl;
    }
    cin.get();
    return 0;
}
```

Sekvenčná verzia pracuje s 1 miliardou parciálnych intervalov. Program meria exekučný čas (v ms), ktorý numerická integrácia alokuje, a vzápätí ho zobrazuje

používateľovi. Na výstup je odosielaná aj vypočítaná hodnota Ludolfovho čísla. Výpočet numerickej integrácie uskutočňujeme celkovo 3-krát.

Po paralelizácii sa zdrojový kód jazyka C++ zmení takto:

```
#include <iostream>
// Import hlavičkového súboru rozhrania OpenMP.
#include <omp.h>
#include <windows.h>

using namespace std;

int main()
{
    double krok, x, obsah = 0.0f;
    long int i, pocetKrokov = 1000000000;
    unsigned int cas0, cas1;
    krok = 1.0f / pocetKrokov;

    for(int n = 1; n <= 3; n++)
    {
        x = obsah = 0.0f;
        cas0 = GetTickCount();
        cout << "Vypocet c. " << n << " bol zahajeny." << endl;
        // Direktíva rozhrania OpenMP na paralelné spracovanie
        // iteratívneho príkazu.
        #pragma omp parallel for reduction(+: obsah) private(x)
        for(i = 0; i < pocetKrokov; i++)
        {
            x = i * krok;
            obsah += 4.0f / (1.0f + x * x);
        }
        cas1 = GetTickCount();
        cout << "Vypocet c. " << n << " bol skonceny." << endl;
        cout.precision(20);
        cout << "Vystup: " << obsah * krok << endl;
        cout << "Cas (ms): " << cas1 - cas0 << endl << endl;
    }
    cin.get();
    return 0;
}
```

Zmeny, ktoré sme do zdrojového kódu zapracovali, sa týkajú dvoch hlavných oblastí:

1. Direktívou predprocesora **#include** zavádzame odkaz na hlavičkový súbor omp.h rozhrania OpenMP.

2. Direktívou kompilátora **#pragma omp parallel for** identifikujeme paralelnú sekciu, ktorá je tvorená iteratívnym príkazom **for**. Ako si môžeme všimnúť, v direktíve aplikujeme klauzuly **reduction** a **private**.

   Klauzula **reduction** sa viaže s premennou **obsah**, v ktorej uchovávame postupne vypočítané obsahy obdĺžnikov, ktorými aproximujeme hodnotu určitého integrálu. Keďže zväzky iterácií cyklu **for** budú spracúvané paralelne, musíme sa vyhnúť potenciálnym pretekom pracovných vlákien, ktoré by viedli k neželaným výsledkom. Povedané inak, je našou povinnosťou zabezpečiť bezpečný priebeh aritmetických operácií, ktoré sú aplikované na premennú **obsah**. Vzhľadom na to, že táto premenná je modifikovaná naprieč rôznymi zväzkami iterácií cyklu, musíme na ňu aplikovať tzv. redukciu. V rámci redukcie sú v paralelne vykonávaných zväzkoch iterácií cyklu vytvorené dočasné lokálne kópie požadovanej premennej, ktoré sú korektne inicializované a modifikované. Po skončení paralelnej exekúcie sú hodnoty všetkých dočasných lokálnych kópií premennej redukované do pôvodnej premennej. Redukčná klauzula používa redukčný operátor **+** s implicitnou nulovou inicializačnou hodnotou dočasných lokálnych kópií premennej.

   Klauzulou **private** explicitne nariaďujeme, že špecifikovaná premenná je súkromná. To znamená, že exekučné prostredie rozhrania OpenMP vytvorí lokálne kópie požadovanej premennej pre každé pracovné vlákno.

Pre úspešné spustenie paralelnej verzie algoritmu numerickej integrácie urobíme v integrovanom vývojovom prostredí produktu Microsoft Visual Studio 2008 toto:

- Ak nie je zobrazené, zviditeľníme podokno **Solution Explorer**.

- Na zdrojový súbor nášho programu jazyka C++, ktorý je umiestnený v priečinku **Source Files**, klikneme pravým tlačidlom myši a z miestnej ponuky vyberieme príkaz **Properties**.

- Po zobrazení dialógového okna **Property Pages** sa zameriame na stromovú štruktúru volieb situovanú na ľavej strane okna.

- V sekcii **Configuration Properties** rozvinieme uzol **C/C++** a klikneme na položku **Language**.

- V zozname napravo vyhľadáme poslednú položku **OpenMP Support** a nastavíme ju na hodnotu **Yes (/openmp)**.

- Dialógové okno **Property Pages** zatvoríme aktiváciou tlačidla **OK**.

- Preložíme zdrojový súbor a zostavíme spustiteľný súbor paralelného programu (**Build → Build Solution**).

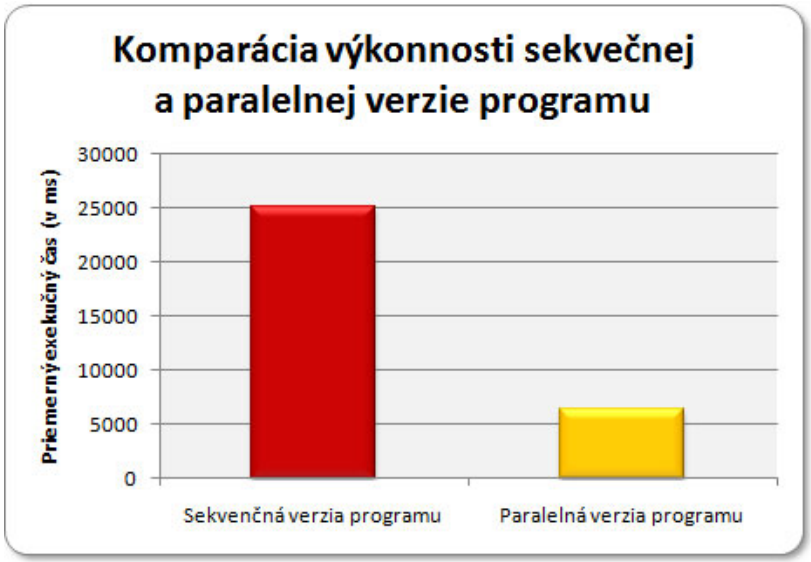
V tab. 18.1 – 18.2 a na obr. 18.6 – 18.7 sú zoskupené výsledky, ktoré sme zaznamenali pri realizácii kolekcií ladiacich (Debug) a ostrých (Release) testov sekvenčnej a paralelnej verzie programu na počítači so 4-jadrovým procesorom Intel Core 2 Quad. Obe kolekcie testov sme uskutočňovali z integrovaného vývojového prostredia produktu Visual Studio 2008. Pri ladiacich testoch sme nepoužívali žiadne optimalizácie. Pri ostrých testoch sme voľbou **Full Optimization (/Ox)** aktivovali úplné optimalizácie.

Kolekcia ladiacich testov (Debug)	Sekvenčná verzia programu (exekučný čas v ms)	Paralelná verzia programu (exekučný čas v ms)
1. výpočet	25054	6349
2. výpočet	25053	6334
3. výpočet	25069	6318
Priemerný exekučný čas	**25059**	**6334**
Nárast výkonu po paralelizácii	**3,96**	

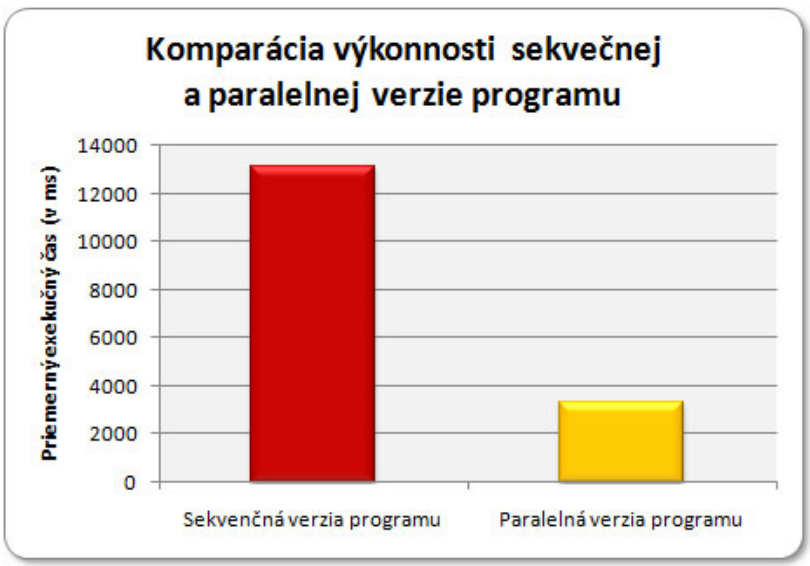
Tab. 18.1: Komparácia výkonnosti sekvenčnej a paralelnej verzie programu realizujúceho numerickú integráciu (kolekcia ladiacich testov)

Kolekcia ostrých testov (Release)	Sekvenčná verzia programu (exekučný čas v ms)	Paralelná verzia programu (exekučný čas v ms)
1. výpočet	13104	3354
2. výpočet	13120	3353
3. výpočet	13104	3354
Priemerný exekučný čas	**13109**	**3354**
Nárast výkonu po paralelizácii	**3,91**	

Tab. 18.2: Komparácia výkonnosti sekvenčnej a paralelnej verzie programu realizujúceho numerickú integráciu (kolekcia ostrých testov)



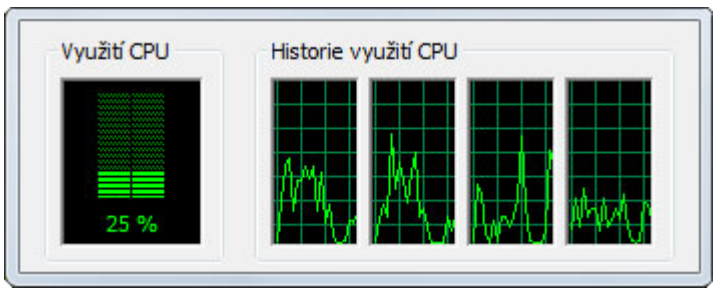
Obr. 18.6: Vizuálna komparácia výkonnosti sekvenčnej a paralelnej verzie programu realizujúceho numerickú integráciu (kolekcia ladiacich testov)



Obr. 18.7: Vizuálna komparácia výkonnosti sekvenčnej a paralelnej verzie

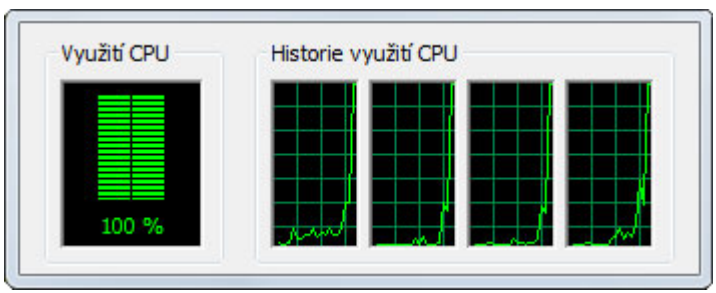
programu realizujúceho numerickú integráciu (kolekcia ostrých testov)

Využitie exekučných jadier viacjadrového procesora sme sledovali pomocou aplikácie **Správca úloh**, ktorá je vstavaná do operačného systému Microsoft Windows Vista. Pri testovaní sekvenčnej verzie programu bolo využívané len jedno exekučné jadro procesora. To je logické, pretože sekvenčná verzia programu obsahuje len jedno (primárne) programové vlákno. Sekvenčná verzia programu využívala len ¼ výpočtovej kapacity počítačového systému vybaveného 4-jadrovým procesorom Intel Core 2 Quad (obr. 18.8).



Obr. 18.8: Sekvenčná verzia programu a jej využitie výpočtových kapacít 4-jadrového procesora

Na druhú stranu, paralelná verzia programu využíva väčšinu exekučného času svojho životného cyklu všetky dostupné exekučné jadrá viacjadrového procesora (obr. 18.9). Je evidentné, že paralelná verzia programu je oveľa efektívnejšia než jej sekvenčný náprotivok.



Obr. 18.9: Paralelná verzia programu a jej využitie výpočtových kapacít 4-jadrového procesora

Záver

Vážené študentky, vážení študenti,

ďakujeme vám za váš záujem o túto knihu. Pevne veríme, že sa jej poradilo splniť vytýčený cieľ a že ste s ňou boli spokojní. Keďže ako vývojári moderného počítačového softvéru sa vzdelávame celý život, dovoľujeme si pripojiť kolekciu odkazov na ďalšie hodnotné informačné zdroje:

1. Centrum paralelného programovania spoločnosti Microsoft → http://msdn.microsoft.com/en-us/concurrency/default.aspx.
2. Visual C# Developer Center → http://msdn.microsoft.com/en-us/vcsharp/default.aspx.
3. C# 3.0 Language Specification → http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/csharp%20language%20specification.doc.
4. Blogs from the C# Team → http://msdn.microsoft.com/en-us/vcsharp/aa336719.aspx.
5. *C# Samples for Visual Studio 2008* → http://code.msdn.microsoft.com/csharpsamples.
6. *Visual C# Technical Articles* → http://msdn.microsoft.com/en-us/vcsharp/bb466181.aspx.
7. *How Do I Videos – Visual C#* → http://msdn.microsoft.com/en-us/vcsharp/bb798022.aspx.
8. *Visual Studio 2008 and .NET Framework 3.5 Training Kit* → http://www.microsoft.com/DOWNLOADS/details.aspx?familyid=8BDAA836-0BBA-4393-94DB-6C3C4A0C98A1&displaylang=en.
9. *MSDN Forums – Visual C#* → http://forums.microsoft.com/MSDN/default.aspx?ForumGroupID=9&SiteID=1.
10. *Webcasty v českom a slovenskom jazyku* → http://www.microsoft.com/cze/msdn/webcasts/default.mspx.

11. *Praktické cvičenia v slovenskom jazyku* → http://www.microsoft.com/slovakia/msdn/hols/default.mspx.
12. *MSDN Magazine* → http://msdn.microsoft.com/en-us/magazine/default.aspx.
13. *C# Corner* → http://www.c-sharpcorner.com/.

Prajeme vám veľa úspechov pri vytváraní tých najlepších počítačových aplikácií v jazyku C# 3.0!

Autor

O autorovi

Ing. Ján Hanák, MVP vyštudoval Ekonomickú univerzitu v Bratislave. Tu, na Katedre aplikovanej informatiky Fakulty hospodárskej informatiky (KAI FHI), pracuje ako vysokoškolský pedagóg. Prednáša a vedie semináre týkajúce sa programovania a vývoja počítačového softvéru v programovacích jazykoch C, C++ a C#. Okrem spomenutej trojice jazykov patrí k jeho obľúbeným programovacím prostriedkom tiež Visual Basic, F# a C++/CLI. Aktívne vynachádza nové postupy tvorby softvéru, ktorý pomáha nielen študentom, ale aj širokej verejnosti.

Je nadšeným autorom odbornej počítačovej literatúry. V jeho portfóliu môžete nájsť nasledujúce knižné tituly:

1. **C#: Akademický výučbový kurz**. Bratislava: Vydavateľstvo EKONÓM, 2009.
2. **Základy paralelného programovania v jazyku C# 3.0**. Brno: Artax, 2009.
3. **Objektovo orientované programovanie v jazyku C# 3.0**. Brno: Artax, 2008.
4. **Inovácie v jazyku Visual Basic 2008**. Praha: Microsoft, 2008.
5. **Visual Basic 2008: Grafické transformácie a ich optimalizácie**. Bratislava: Microsoft Slovakia, 2008.
6. **Programovanie B – Zbierka prednášok (Učebná pomôcka na programovanie v jazyku C++)**. Bratislava: Vydavateľstvo EKONÓM, 2008.
7. **Programovanie A – Zbierka prednášok (Učebná pomôcka na programovanie v jazyku C)**. Bratislava: Vydavateľstvo EKONÓM, 2008.
8. **Expanzívne šablóny: Príručka pre tvorbu "code snippets" pre Visual Studio**. Bratislava: Microsoft Slovakia, 2008.
9. **Kryptografia: Príručka pre praktické odskúšanie symetrického šifrovania v .NET Framework-u**. Bratislava: Microsoft Slovakia, 2007.
10. **Príručka pre praktické odskúšanie vývoja nad Windows Mobile 6.0**. Bratislava: Microsoft Slovakia, 2007.

11. **Príručka pre praktické odskúšanie vývoja nad DirectX**. Bratislava: Microsoft Slovakia, 2007.
12. **Príručka pre praktické odskúšanie automatizácie aplikácií Microsoft Office 2007**. Bratislava: Microsoft Slovakia, 2007.
13. **Visual Basic 2005 pro pokročilé**. Brno: Zoner Press, 2006.
14. **C# – praktické příklady**. Praha: Grada Publishing, 2006 (kniha získala ocenenie „Najúspešnejšia novinka vydavateľstva Grada v oblasti programovania za rok 2006“).
15. **Programujeme v jazycích C++ s Managed Extensions a C++/CLI**. Praha: Microsoft, 2006.
16. **Přecházíme z jazyka Visual Basic 6.0 na jazyk Visual Basic 2005**. Praha: Microsoft, 2005.
17. **Visual Basic .NET 2003 – Začínáme programovat**. Praha: Grada Publishing, 2004.

V rokoch 2006, 2007 a 2008 bol jeho prínos vývojárskym komunitám ocenený celosvetovými vývojárskymi titulmi **Microsoft Most Valuable Professional (MVP)** s kompetenciou **Visual Developer – Visual C++**.

Spoločnosť Microsoft udelila Ing. Jánovi Hanákovi, MVP v roku 2009 **ocenenie za zlepšovanie akademického ekosystému** a **za signifikantné rozširovanie technológií a programovacích jazykov** Microsoftu na akademickej pôde.

Kontakt s vývojármi a programátormi udržiava najmä prostredníctvom technických seminárov a odborných konferencií, na ktorých vystupuje. Za všetky vyberáme tieto:

- Konferencia **Software Developer 2007**, príspevok na tému „Představení produktu Visual C++ 2005 a jazyka C++/CLI“. Praha 19. 6. 2007.

- Technický seminár **Novinky vo Visual C++ 2005**. Microsoft Slovakia. Bratislava 3. 10. 2006.

- Technický seminár **Visual Basic 2005 a jeho cesta k Windows Vista**. Microsoft Slovakia. Bratislava 27. 4. 2006.

Ako autor má mnohoročné skúsenosti s prácou v elektronických a printových médiách. Počas svojej kariéry pôsobil ako odborný autor alebo odborný redaktor v nasledujúcich časopisoch: PC WORLD, SOFTWARE DEVELOPER, CONNECT!, COMPUTERWORLD, INFOWARE, PC REVUE a CHIP.

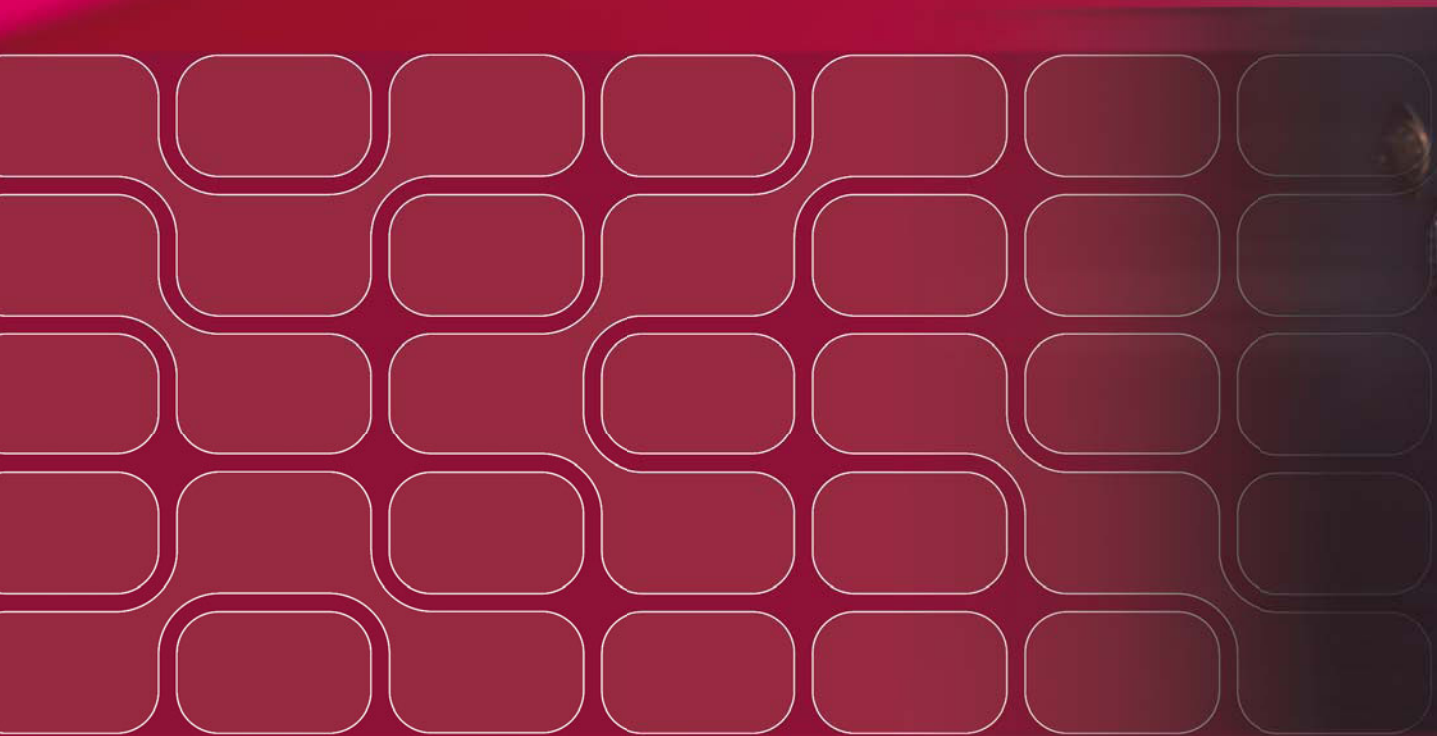
Dovedna publikoval viac ako 250 odborných a populárnych prác venovaných vývoju počítačového softvéru.

Akademický blog autora môžete sledovať na adrese: http://blog.aspnet.sk/hanja/.

Ak sa chcete s autorom spojiť, môžete využiť jeho adresu elektronickej pošty: hanja@stonline.sk.

Použitá literatúra

1. Akhter, S., Roberts, J.: Multi-core Programming. Hillsboro: Intel Press, 2006.
2. Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., White, A.: Sourcebook of Parallel Computing. San Francisco: Morgan Kaufmann Publishers, 2003.
3. Hanák, J.: Objektovo orientované programovanie v jazyku C# 3.0. Brno: Artax, 2008.
4. Hanák, J.: Inovácie v jazyku Visual Basic 2008. Praha: Microsoft, 2008.
5. Hanák, J.: Programujeme v jazycích C++ s Managed Extensions a C++/CLI. Praha: Microsoft, 2006.
6. Hanák, J.: C# – praktické příklady. Praha: Grada Publishing, 2006.
7. Stoecker, M.: Developing Windows-based Applications with Microsoft Visual Basic .NET and Microsoft Visual C# .NET, 2002.
8. Pratschner, S.: Customizing the Microsoft .NET Framework Common Language Runtime. Washington: Microsoft Press, 2005.



Ing. Ján Hanák je Microsoft MVP (Most Valuable Professional – najcennejší odborník) s kompetenciou Visual Developer – Visual C++. Je autorom 17 odborných kníh, príručiek a praktických cvičení o programovaní a vývoji počítačového softvéru. Pracuje ako vysokoškolský pedagóg na Katedre aplikovanej informatiky Fakulty hospodárskej informatiky Ekonomickej univerzity v Bratislave. Prednáša a vedie semináre z programovania v jazykoch C, C++ a C#. V rámci svojej vedeckej činnosti sa zaoberá problematikou štruktúrovaného, objektovo orientovaného, komponentového, funkcionálneho a paralelného programovania.

ISBN: 978-80-87017-03-6