JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH

Pedagogická fakulta

DIPLOMOVÁ PRÁCE

2007

Karel Attl

Karel Attl

Tvorba vláknových aplikací v jazyce Java

JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH

Pedagogická fakulta

Katedra informatiky

Tvorba vláknových aplikací v jazyce Java

Diplomová práce

Autor: Karel Attl

Vedoucí diplomové práce: Ing. Václav Novák, CSc.

ANOTACE

Tato diplomová práce pojednává o programování vícevláknových aplikací v jazyce Java. S verzí Java 5 se v API jazyka objevuje i knihovna `java.util.concurrent`, která významným způsobem ulehčuje a zefektivňuje návrh paralelních aplikací. Práce je pojatá jako úvod do programování vícevláknových aplikací, a zároveň ji lze využít jako studijní materiál. Teoretický úvod pojednává o procesech a technologickému pozadí multitaskingu jako analogie k vláknům, zároveň se dotýká technologie Java a pozadí práce s pamětí. Zbytek diplomové práce už se věnuje praktickému nastínění práce s vlákny. Pokrývá toto téma od úplných základů, jakými je vytvoření objektu typu `Thread`, přes pokročilejší úlohy, jakými je práce s balíčkem `java.util.concurrent` a na závěr věnuje prostor problémům, se kterými se programátor může setkat při vývoji vícevláknových aplikací.

ABSTRACT

This diploma thesis is aimed at programming of multithreaded applications in Java. With Java 5 comes package `java.util.concurrent`, which in an important way makes developing of parallel applications easier and more effective. This work is conceived as an introduction to programming of multithreaded applications in Java and could be also used as an educational material. Theoretical introduction about processes and technological background of multitasking gives analogy to threads, at the same time it is touching on Java technology and how Java works with memory. The rest of this diploma thesis concerns practical work with threads. This topic is covered from absolute beginning, which means creating `Thread` objects, including advanced topics like working with package `java.util.concurrent` and also some problems that can appear when writing multithreaded applications.

P r o h l á š e n í

Prohlašuji, že svoji diplomovou práci na téma

„Vývoj vláknových aplikací v jazyce Java“

jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své diplomové práce, a to v nezkrácené podobě – v úpravě vzniklé vypuštěním vyznačených částí archivovaných Pedagogickou fakultou elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách.

....

Karel Attl

V Českých Budějovicích, dne ……………………..

Poděkování

Chtěl bych touto cestou poděkovat RNDr. Jaroslavu Ichovi a Ing. Václavu Novákovi, CSc., za cenné rady a poskytnuté materiály, které mi umožnily sepsat tuto práci.

Obsah

1 Úvod do vícevláknového programování .... - 10 -

2 Multithreading, multitasking, procesy a vlákna .... - 13 -

2.1 Úvod do multihtreadingu .... - 13 -
2.2 Operační systémy a procesy .... - 15 -
2.2.1 Operační systém .... - 15 -
2.2.2 Proces .... - 17 -
2.2.3 Správa paměti operačním systémem .... - 18 -
2.2.4 Změna kontextu – přepínání mezi procesy .... - 19 -
2.2.5 Blokové schéma stavů procesu .... - 20 -
2.2.6 Techniky plánování CPU .... - 21 -
2.2.7 Algoritmy plánování čekajících procesů .... - 22 -
2.2.8 Meziprocesová komunikace .... - 23 -
2.2.9 Některé důležité pojmy související s procesy .... - 24 -
2.3 Princip multitaskingu .... - 25 -
2.3.1 Kooperativní multitasking .... - 26 -
2.3.2 Preemptivní multitasking .... - 26 -
2.3.3 Implementace multitaskingu .... - 26 -
2.4 Vlákno .... - 28 -

3 Úvod do paralelního programování v jazyce Java .... - 29 -

3.1 Krátká historie technologie Java .... - 30 -
3.2 Filosofie Javy .... - 31 -
3.2.1 Platforma Java .... - 31 -
3.3 Programovací jazyk Java .... - 34 -
3.4 Java Memory Model .... - 35 -

4 Základy práce s vlákny v jazyce Java .... - 38 -

4.1 Objekty typu Thread .... - 39 -
4.1.1 Životní cyklus vlákna .... - 40 -
4.1.2 Priority vlákna .... - 41 -
4.2 Vytvoření vlákna .... - 42 -
4.2.1 Vytvoření vlákna pomocí rozhraní Runnable .... - 43 -
4.2.2 Vytvoření vlákna odděděním od třídy Thread .... - 43 -
4.3 Pozastavení běhu vlákna .... - 44 -
4.4 Přerušení běhu vlákna .... - 46 -
4.4.1 Shrnutí – projekt Práce s vlákny .... - 46 -
4.5 Úvod do synchronizace vláken .... - 47 -
4.5.1 Interference vláken .... - 47 -
4.5.2 Chyby v konzistenci paměti .... - 48 -
4.6 Synchronizace .... - 50 -
4.6.1 Vnitřní zámky a monitor, fungování synchronizace .... - 50 -
4.6.2 Synchronizované metody .... - 51 -
4.6.3 Synchronizované příkazy .... - 52 -
4.6.4 Atomicita a klíčové slovo volatile .... - 53 -
4.6.5 Neměnné (immutable) objekty .... - 54 -
4.7 Problémy s aktivitou .... - 55 -
4.7.1 Zablokování (deadlock) .... - 55 -
4.7.2 Odepření zdrojů (starvation) .... - 56 -
4.7.3 Vzájemné brzdění (livelock) .... - 56 -

5 Knihovna java.util.concurrent ... - 57 -
5.1 Atomické proměnné ... - 57 -
5.1.1 Práce s atomickými proměnnými ... - 58 -
5.1.2 Atomická pole ... - 60 -
5.1.3 Atomické změny členských proměnných třídy ... - 61 -
5.2 Rozhraní Lock ... - 61 -
5.2.1 Výhody oproti synchronized bloku ... - 62 -
5.2.2 Reentrantní a nereentrantní zámky ... - 63 -
5.3 Synchronizers (Synchronizační primitiva) ... - 64 -
5.3.1 Typ „semafor“ ... - 64 -
5.3.2 Typ bariéra ... - 65 -
5.3.3 Exchanger ... - 66 -
5.4 Exekutory ... - 66 -
5.4.1 Rozhraní Executor ... - 67 -
5.4.2 Rozhraní ExecutorService ... - 68 -
5.4.3 Rozhraní ScheduledExecutorService ... - 69 -
5.4.4 Skupiny vláken (Thread Pool) ... - 70 -
5.5 Paralelní kolekce ... - 73 -
5.5.1 Fronta (Queue) ... - 74 -
5.5.2 Seznam (List) ... - 78 -
5.5.3 Slovník/Mapa (Map) ... - 78 -
5.5.4 Množina (Set) ... - 80 -

6 Testování a výkon vícevláknových aplikací ... - 82 -
6.1 Problémy s aktivitou ... - 82 -
6.1.1 Deadlock ... - 82 -
6.1.2 Ostatní selhání aktivity ... - 86 -
6.2 Zlepšování výkonu ... - 87 -
6.2.1 Filosofie výkonných aplikací ... - 88 -
6.2.2 Monitorování výkonu CPU ... - 89 -
6.2.3 Shrnutí ... - 91 -
6.3 Testování vícevláknových programů ... - 91 -
6.3.1 Testování na správnost ... - 92 -
6.3.2 Testování na výkon ... - 92 -
6.3.3 Profilery a monitory ... - 93 -
6.3.4 Nejčastější chyby vícevláknových programů ... - 94 -

7 Závěr ... - 95 -

1 Úvod do vícevláknového programování

S postupným a zároveň velmi rychlým vývojem počítačového odvětví dochází i ke stále vyšším požadavkům na výkon systémů. Moderní výpočetní systémy, ale i osobní počítače, zařízení PDA či mobilní telefony jsou stále dokonalejší, a spolu se stále se zvyšujícími možnostmi těchto zařízení stoupá i jejich výkonnost a schopnost zpracovávat mnohem více informací za kratší dobu. Stejně tak se ale pomalu naráží na technologický strop a fyzikální limity, je tedy potřeba začít využívat jiné myšlenky a postupy, které dokáží vyhovět požadavkům na vysoký výkon a efektivnější využívání zdrojů. Objevují se tak technologie, které mají tvorbu a návrh aplikací urychlit, zjednodušit a zároveň co nejvíce zefektivnit.

Jednou z nejvýznamnějších technologií je i užití tzv. **vláken.** Vlákna dokáží simulovat či částečně nahradit víceúlohovost operačního systému i v případě, že není navržen tak, aby to hardwarově umožňoval. Pokud ano, dokáží ještě více urychlit běh programu a zároveň optimálně využívat systémové prostředky k co největšímu výkonu. Vláknové programování a knihovny pro práci s vlákny se tak staly velmi důležitou součástí Javovské platformy. A stejně tak, jako se v praxi stále více prosazují vícejádrové procesory, nelze si představit výkonné podnikové aplikace a rozsáhlé systémy jinak, než založené na vícevláknových programech a tzv. konkurenčním programování. Java 5 posunula tuto technologii o velký kus dále a silně vylepšila práci s vlákny oproti starším verzím tak, že vhodně navržené programy převede Java Virtual Machine na vysoce výkonné aplikace. Knihovna `java.util.concurrent` zároveň významným způsobem usnadňuje řešení některých typických problémů při práci s vlákny.

Jaké jsou tedy největší výhody a klady přístupu založeného na vláknovém programování? Objekty, jakožto základ objektově orientovaného programování, nám umožňují rozdělit program na jednotlivé logické úseky. To

ale nemusí vždy stačit a my potřebujeme program rozdělit i na samostatné a na sobě nezávislé úlohy. Toho dosáhneme pomocí vláken. Na vlákno se můžeme dívat tak, jako by bylo samostatným „podprogramem“ a mělo k dispozici vlastní CPU. Víceúlohový operační systém obsahuje mechanismy, které se starají o rozdělování času a výkonu procesoru tak, že programátora tato záležitost nemusí zajímat. Operační systém je tak schopen spouštět úlohy v rychlém sledu za sebou tak, že z pohledu uživatele to vypadá, že běží současně.

Největší výhoda je tedy zřejmá – můžeme vytvářet program, jehož jednotlivé části jsou na sobě de facto nezávislé. Taková situace nastane velmi často – například pokud program vykonává nějakou výpočetně a časově náročnou úlohu, nemusí zbytek programu čekat na dokončení této úlohy a my se nedostaneme do pozice, kdy program tzv. zatuhne, nereaguje, a my musíme počkat na dokončení úlohy, než zase bude možné program ovládat. Vytvoříme tedy vlákno, které bude mít na starosti pouze tuto výpočetně náročnou úlohu, a ta pak běží nezávisle na hlavním programu. Nejjednodušším příkladem, kde použít vlákna, je tlačítko „Ukončit program“. Určitě není vhodné koncipovat toto tlačítko tak, že kód pro obsluhu tlačítka vložíme do každého myslitelného bloku programu jen proto, aby na jeho stisknutí bylo možné kdykoliv zareagovat. Inteligentnější a šetrnější řešení je umístit obsluhu tlačítka do vlastního vlákna, a to tak je přístupné kdykoliv a nezávisle na tom, co právě náš program dělá. Jedním z hlavních důvodů pro tvorbu vláknových aplikací je tak „user-friendly“ (uživatelsky přívětivé) prostředí programu s grafickým uživatelským rozhraním.

Tato práce si klade za cíl vysvětlit, jak co nejefektivněji používat vlákna v jazyce Java, vysvětlit, jak operační systém a platforma s nimi pracuje a zároveň nastínit či úplně rozvést co nejoptimálnější řešení těch nejzákladnějších úloh a situací, kde se lze s vlákny setkat. Zároveň se nevyhneme problémům, které mohou vzniknout na úrovni programovacího jazyka, a ukážeme jak jim předejít a řešit. Je však jasné, že v této práci nelze

zmínit všechny aspekty a techniky, které se dotýkají vláknového programování v Javě.

Dále tato práce zahrnuje popis nejdůležitějších tříd a rozhraní balíčku `java.util.concurrent` a vzhledem k více než dostatečné dokumentaci jsem se zaměřil spíše na řešení či nastínění konkrétních problému s využitím těchto tříd a rozhraní. Práce je ve značné míře proložena jak funkčními programy, tak bloky kódu, které názorně ukazují nejzásadnější vlastnosti a slouží k lepšímu pochopení situace. Pro programování aplikací používám vývojová prostředí BlueJ a NetBeans BlueJ Edition, součástí této práce je i CD obsahující ukázkové programy a projekty.

Zároveň se pokusím se využít zdrojů na internetu a monografií, věnujících se vláknovému programování v jazyce Java 5, a zpracovat tuto tématiku i jako vhodný studijní materiál.

2 Multithreading, multitasking, procesy a vlákna

2.1 Úvod do multihtreadingu

Multithreading (překládáno jako vícevláknovost, já však budu používat výraz multithreading, popřípadě zkratku MT) je technika umožňující operačnímu systému obsluhovat více úloh v podstatě souběžně. Základní koncept vznikl před několika desetiletími ve vývojových laboratořích a výzkumných centrech. Myšlenka byla realizována mimo jiné v jazycích Concurrent Pascal nebo Ada. Ovšem k obrovskému rozvoji došlo až v průběhu devadesátých let dvacátého století, spolu s vývojem moderních desktopových, ale i jiných operačních systémů.

V dnes už vzdálené minulosti počítače nefungovaly na principu, na jakém je známe dnes. Spouštěly jednotlivé programy podle principu „*from beginning to end*“ (Program byl čten po řádcích od začátku do konce, stejně jako kniha. Co se vyskytlo před aktuálně čteným místem už program „znal“, příkazy které následovaly, byly přístupny až v okamžiku čtení) a běžící program měl výhradní přístup ke všem systémovým zdrojům. Každý proces byl v podstatě virtuální von Neumannův stroj s vlastním paměťovým prostorem, kde byla uložena jak data, tak instrukce. Spuštění probíhalo sekvenčně v závislosti na jazyce a sémantice daného stroje. Pro každou instrukci zároveň existovala její instrukce následující a program byl vykonáván jako sekvence příkazů daného jazyka. Běh všech programů tedy byl založen na spouštění instrukce, která následuje po právě vykonané instrukci.

Výhoda tohoto přístupu spočívala v tom, že každý program mohl využívat systémové zdroje na 100% a nemusel se o ně s jiným procesem dělit. Nevýhody ovšem jsou také zřejmé a převažují. Tou hlavní je silná neefektivita. Jak časová tak ekonomická – neefektivní využívání zdrojů.

Model sekvenčního programování je pro člověka intuitivní a přirozený, v podstatě kopíruje lidskou práci. Člověk také vykonává pouze jednu činnost v jeden okamžik, celý život se učí algoritmické postupy, které vedou k výsledku. Každá jeho činnost je posloupností úkonů, které musí člověk vykonat, aby dosáhl daného cíle. Ne vždy ale stačí vykonávat pouze jednu věc v jeden okamžik, často je to nežádoucí a hlavně neefektivní. Uveďme si jeden příklad za všechny, takový, jaký všichni známe. Ranní vstávání. Člověk obvykle vstává na poslední chvíli, a součet časů událostí potřebných pro přípravu na odchod z domu je obvykle delší než čas, který doopravdy má. Je tedy nutné udělat několik věcí najednou. Je tak více než vhodné přerozdělit čas a úkoly tak, aby vše zvládl. Pokud dá vařit vodu na kávu a bude čekat, než se uvaří, nestihne nic. Mezitím, než se mu uvaří voda, si však může připravit snídani, osprchovat se a umýt, a zalije kávu ve chvíli kdy už má některé další věci hotovy. Na tomto modelu můžeme pochopit, jak procesor asynchronně obsluhuje ostatní úlohy a podle jejich časové či výpočetní náročnosti jim inteligentně přiděluje systémové zdroje, zatímco jiný proces na něco čeká.

Operační systémy se tak postupně vyvinuly k možnosti spouštět několik programů ve stejný okamžik. K tomu operační systém využívá tzv. **procesy**. Proces umožní spustit program izolovaně od dalších běžících programů, nezávisle na ostatních programech jsou mu operačním systémem přiděleny zdroje, alokována paměť, přístup k souborovým systémům a je ošetřeno zabezpečení procesu. Procesy sice běží odděleně, ale bylo by velkým mínusem této myšlenky, kdyby spolu nedokázaly komunikovat. To je zajištěno různými technologiemi jako např. sockety, signály, sdílená paměť, semafory, či soubory do kterých mohou procesy zapisovat a sdílet mezi sebou data.

Multithreading je v podstatě přenesení multitaskingu a jeho aplikace do programovacího jazyka. Stejně tak jako operační systém obsluhuje několik úloh najednou, tak interpret jazyka zabezpečuje běh několika vláken, která jsou analogií k procesům v hierarchii programovacího jazyka. Tím se může

nejen zrychlit běh programu, ale zároveň se stává (zejména časově) více efektivním.

Myšlenka multithreadingu sebou nese několik zásadních vlastností oproti klasickému jednoprogramovému běhu. Ty nejdůležitější jsou:

- **Inteligentní využití zdrojů**
  Programy často čekají na externí operaci či událost (např. vstup) – v takovém okamžiku jsou systémové zdroje nevyužity. Je efektivní přidělit je tedy jinému programu.

- **Spravedlivé rozdělení zdrojů**
  Všem programům je přidělován procesorový čas a výkon zhruba stejně spravedlivě. Záleží samozřejmě na potřebách jednotlivých procesů, jejich důležitosti či náročnosti.

- **Zjednodušení návrhu aplikace**
  Často je mnohem jednodušší navrhnout několik podprogramů, z nichž každý vykonává jednu konkrétní úlohu a koordinace těchto programů, než vytvářet složitější program, který zvládá všechno.

2.2 Operační systémy a procesy

K tomu, abychom pochopili, jak vlastně funguje multithreading, potřebujeme si říci něco o tom, jak operační systém pracuje s procesy a jakým způsobem je řešen multitasking. Multithreading je pak analogií multitaskingu přenesenou do prostředí programovacího jazyka.

2.2.1 Operační systém

Na operační systém můžeme pohlížet jako na balík programů („systémový software“), které umožňují co nejefektivnější využívání hardwaru. Hlavním

úkolem operačního systému je zprostředkovat a zabezpečit běh programů („aplikační software"). Operační systém zaštiťuje různé verze a implementace hardwaru tak, aby pro program, který využívá služeb operačního systému, byly jednotlivé prvky hardwaru transparentní. To v podstatě znamená, aby aplikace používala jednotný přístup ke službám operačního systému nezávisle na použitém hardware. Dále poskytuje operační systém různé služby, které podporují snazší implementaci aplikačních programů, např. služby souborového systému, síťové služby, apod.

Nejpoužívanějšími platformami jsou v současné době operační systémy Windows od firmy Microsoft (nejčastěji ve verzi XP) a operační systémy UNIXového typu (Linux). Liší se jak samotnou filosofií operačního systému, tak ve způsobech práce s daty či konfiguraci.

Základní úkoly operačního systému:

- řízení procesů a přidělování prostředků
- správa souborového systému
- správa paměťových datových médií
- správa textových a grafických rozhraní
- interakce s uživatelem
- správa síťových rozhraní
- správa vstupně výstupních rozhraní (USB, LPT, PS2, …)
- správa multimediálních rozhraní
- správa ovladačů zařízení

Operační systém pro osobní počítače (PC) je rozdělen na dvě části:

- **BIOS** (Basic Input Output System)
  BIOS je program uložený v paměti ROM (nebo nahrán v paměti (E)EPROM) jako firmware a slouží k tomu, aby počítač po zapnutí provedl korektní inicializaci periférií a zavedl operační systém.

- **Vlastní operační systém**
  Je program, který je BIOSem nahrán z diskety, pevného disku nebo jiného média. Úkolem tohoto programu je poskytovat uživatelům a vývojářům aplikací jednotný přístup zejména k datům na disku, k síti a ke vstupním a výstupním zařízením. Moderní operační systémy umožňují multitasking a obsahují velmi vyvinutou a konfigurovatelnou podporu ostatních programů. Zároveň nejsou už jen prostředím pro běh aplikací třetích stran, ale jsou komplexním balíkem aplikací a utilit, které se snaží uživateli co nejvíce ulehčit práci s počítačem.

Nás bude v této práci zajímat, jakým způsobem pracuje desktopový operační systém na PC s pamětí a procesy a jak díky tomu funguje multitasking a multithreading.

2.2.2 Proces

Proces definujeme jako program spuštěný v operační paměti. Programem rozumíme sled příkazů a instrukcí, které jsou převedeny z programovacího jazyka do podoby, která je srozumitelná vykonavateli programu (interpret jazyka, operačním systém). Proces je pak tento program zavedený do operační paměti a obsluhovaný procesorem. Obsahuje nejen kód programu, ale i data, která se mohou dynamicky měnit v závislosti na běhu programu a životním cyklu procesu.

Proces používá systémové prostředky počítače, které mu přidělil operační systém. Většina těchto informací je uložena ve speciální datové struktuře jádra operačního systému, která se nazývá PCB (Process Control Block).

Uložené informace v PCB například zahrnují:

- spustitelný kód programu umístěný v souboru na disku
- paměť, do které je program umístěn (tj. strojové instrukce nebo jejich část)
- paměť pro zásobník a data, která proces zpracovává
- prostředí procesu (otevřené soubory, proměnné prostředí)
- bezpečnostní informace o procesu (vlastník, oprávnění apod.)
- přidělené systémové prostředky
- stav procesoru (uchovávaný kvůli změně kontextu)

2.2.3 Správa paměti operačním systémem

Správa paměti (memory management) znamená soubor úkonů a opatření, které souvisí s hospodařením s operační pamětí počítače. V té nejjednodušší formě zahrnuje pouze přidělování paměti jednotlivým procesům. Zajišťuje však i následné uvolňování paměti (například pokud je proces ukončen nebo potřebuje méně paměti), nastavuje ochranu paměti a eventuelně i správu adresování paměti. O všechny tyto činnosti se v počítači stará tzv. **modul správy paměti**, což je součást operačního systému. Pro některé z výše uvedených úkolů je potřeba hardwarová podpora uvnitř procesoru.

Současná architektura počítačů využívá hierarchické uspořádání paměti – od nejrychlejších registrů přes cache procesoru a paměť s náhodným přístupem (random access memory) až po data uložená na pevných discích. Správce paměti operačního systému koordinuje použití těchto různých druhů paměti tak, že hledá která je dostupná, určuje, kterou alokovat či dealokovat a jak přesouvat data mezi nimi. Tato aktivita, obvykle nazývaná virtuální správa paměti (Virtual Memory Management), zvyšuje objem paměti tím, že hledá místo i na jiných typech paměti a využívá je jako odkládací paměť. Samozřejmě, že takto vzniká i jistý rychlostní deficit, protože tato média jsou pomalejší než RAM. Pokud dojde k tomu, že běžící proces vyžaduje více paměti než je dostupná, dochází k tzv. swapování a běh procesů se tak

zpomalí. Operační systém také „odkládá“ neaktivní stránky v paměti na nějaké pomalejší médium a uvolňuje tak rychlejší paměť pro běžící procesy. Tento úkon nazýváme swapování nebo také stránkování.

Další důležitou úlohou pro správce paměti je správa virtuálních adres. Pokud je spuštěno více procesů, je úkolem správce zabezpečit je tak, aby nedocházelo k jejich zasahování do paměti jiných procesů (výjimkou je samozřejmě požadavek na sdílenou paměť). To je zajištěno oddělenými paměťovými prostory. Každý proces vidí celý virtuální paměťový prostor, typicky od adresy 0 až po maximální velikost paměti. Operační systém pak pomocí stránkovací tabulky převádí virtuální adresy na fyzické. Paměťové prostory jsou alokovány až do okamžiku ukončení procesu, pak je jeho paměť uvolněna pro jiný proces.

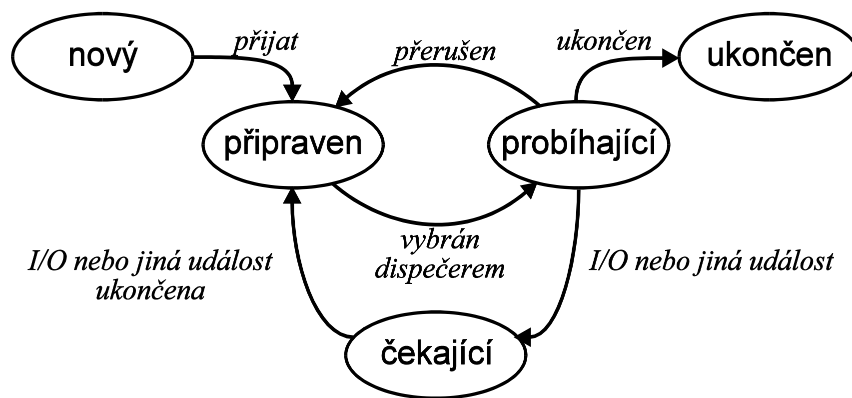
2.2.4 Změna kontextu – přepínání mezi procesy

Moderní operační systémy umožňují spustit v jeden okamžik více procesů (multitasking), takže je možné spustit i jeden program vícekrát, čímž vznikne několik různých procesů, které se navzájem liší. Pokud je v počítači méně procesorů, než je běžících procesů, musejí procesory nějakým způsobem tuto situaci ošetřit a procesy střídat. Tato změna obsluhovaného procesu se nazývá **změna kontextu** (Context switch).

Pod slovem kontext si můžeme představit aktuální stav procesu ve chvíli, kdy dochází ke změně kontextu. Změna kontextu je tedy operace, při níž operační systém přepne řízení mezi procesy. Dojde k uložení současného procesu a zároveň spuštění dalšího. Tento děj se u moderních procesorů opakuje mnohokrát za vteřinu. Změny kontextu jsou obvykle výpočetně intenzivní a silně optimalizované a plánované úlohy, k čemuž jsou operační systémy přizpůsobeny. Změnou kontextu můžeme myslet nejen změnu kontextu procesů, ale i registrů nebo vláken.

2.2.5 Blokové schéma stavů procesu

Každý proces má pět základních stavů a dva rozšířené, přičemž v aktuálním okamžiku se může nacházet právě v jednom z nich.



Obr. 1 – Blokové schéma stavů procesů

Základní stavy procesů

- **Nový**
  Proces je vytvořen příkazem uživatele, na pokyn operačního systému, či na žádost jiného procesu (rodičovského procesu)
- **Připraven**
  Proces je připraven pro vstup do stavu probíhající, čeká pouze na přidělení procesoru

- **Probíhající**
  Procesu je přidělen procesor a vykonávají se příslušné instrukce
- **Čekající (blokovaný)**
  Proces čeká na I/O operaci, skončení jiného procesu, či uvolnění zdroje
- **Ukončen**

Rozšířené stavy procesů

Tyto stavy jsou dostupné pouze v systémech, které podporují virtuální paměť. V obou stavech jsou procesy uloženy v sekundární paměti (typicky pevný disk)

- **Odložený a blokovaný**
  Proces je sice evidován jako připravený, ale je odložen a blokován a čeká na úplné uvolnění systémových zdrojů.
- **Odložený a čekající**
  Proces je také odložen do virtuální paměti, ale nečeká až na úplné vyprázdnění fronty čekajících procesů a může být kdykoliv převeden do stavu "připraven".

2.2.6 Techniky plánování CPU

Zmíníme zde dvě techniky, které procesor používá pro plánování procesů:

- **Nepreemptivní plánování**
  Proces opouští procesor pouze v okamžiku, kdy čeká nebo je ukončen, tedy dobrovolně
- **Preemptivní plánování**
  Plánovač procesoru přiřazuje každé úloze pouze určitý čas. Jestliže pak

proces neodevzdá procesor další úloze, je ukončen. To zaručí, že procesor není stále obsazen jedním procesem, pokud jiné procesy čekají.

2.2.7 Algoritmy plánování čekajících procesů

Zde zmíníme několik různých přístupů a postupů v plánování a obsluze čekajících procesů. Tyto postupy a algoritmy lze aplikovat i na vlákna a jejich pochopení nám usnadní i práci s vlákny.

FCFS – First comes, first served

Algoritmus FCFS je vlastně implementací fronty (FIFO – First in, first out) kdy první proces který přijde je obsloužen procesorem. U tohoto algoritmu můžeme předpokládat jeho velmi nízkou efektivitu a dlouhé čekací doby, z čehož plyne velmi nízký výkon. Pokud proces obdrží procesor, drží si jej tak dlouho, dokud není ukončen. Vznikají tak velmi často tzv. convoy effects, kdy kratší procesy čekají na obsloužení dlouhého. Toto plánování je nepreemptivní.

Cyklická obsluha

Plánování, při kterém má proces přidělenu pevnou časovou jednotku, po jejímž uplynutí dochází ke změně kontextu a přepnutí na další čekající proces.

Plánování pomocí více front

Procesy jsou rozděleny do skupin (front) podle své povahy, podle čehož jsou taky dále obsluhovány. Každá fronta je plánována s jinou prioritou a jiným algoritmem. Dochází k plánování podle priority, proces z „nižší“ skupiny nemůže být obsloužen, pokud není „vyšší“ fronta prázdná.

SJF – Shortest job first

Tento algoritmus pracuje tak, že s každým procesem je asociován čas potřebný na jeho vykonání. Proces, který má nejkratší čas potřebný pro dokončení, bude vykonán nejdříve. Pokud se objeví dva procesy se stejným časem, dochází k rozhodování pomocí FCFS.

Plánování podle priority

Proces má přidělenu hodnotu, prioritu, podle které jsou pak vybírány obsluhované procesy. Procesy s vyšší prioritou tak jsou obsluhovány dříve. Pokud mají dva nebo více procesů stejnou prioritu, rozhoduje se opět podle FCFS.

Rozlišujeme prioritu interní a externí. Interní priorita znamená nějakou měřitelnou hodnotu odvozenou ze samotného procesu – paměťové nároky, počet obsluhovaných souborů apod. Externí priorita je nastavována podle nějakých vnějších kritérií, je pevně daná nezávisle na kritériích pro prioritu externí.

2.2.8 Meziprocesová komunikace

Dalším okruhem, který považuji za vhodné alespoň okrajově zmínit, je to, jakým způsobem mezi sebou procesy komunikují. Procesy samozřejmě mohou běžet jeden vedle druhého, ale mnohdy je vhodnější jejich ať už těsná nebo volnější kooperace.

To, čím nazýváme meziprocesovou komunikaci, rozumíme souhrn funkcí, pravidel a operací, které umožňují procesům vzájemnou interakci.

Ty nejzákladnější prostředky meziprocesové komunikace jsou:

- Spuštění nových (nezávislých) procesů
- Vytvoření podprocesu
- Vzájemná komunikace pomocí rour (pipe – v podstatě fronta zpráv)
- Komunikace pomocí signálů – signál je zpráva zasílaná jiným procesem (popř. výjimečně jádrem operačního systému), která slouží k informování procesu, že došlo k nějaké události. Vykonávání procesu se při přijímání signálu pozastaví. Signál umožňuje vykonávat nad procesem různé události – např. *exit, ignore, stop, continue*.
- Alokace sdílené paměti
- Obsluha událostí pomocí semaforu – celočíselný čítač, který obsluhuje procesy a vpouští je do procesoru. Funguje podobně jako semafor na železnici, nastavuje pro procesy volno a stop. Slouží k ošetření volných prostředků.

2.2.9 Některé důležité pojmy související s procesy

Synchronizační primitivum

Synchronizační primitivum je prostředek umožňující zároveň běžícím aplikacím ošetřit současný přístup ke sdíleným prostředkům. Z těch nejdůležitějších zmíníme monitor, semafor a frontu zpráv. Všechny pracují na principu uzamykání prostředků pro proces. Někdy je v režii operačního systému, jindy platformy programovacího jazyka. Chybné použití synchronizačních primitiv vede k chybám, kdy více procesů přistupuje k jednomu prostředku a dochází ke kolizi – vzniká tzv. deadlock.

Deadlock

Deadlock je výraz pro situaci, kdy dochází k zacyklení procesu. Tzn., že proces A čeká na dokončení procesu B, proces B ale nemůže být ukončen, dokud nebude ukončen proces A. Vzniká tak nekonečná smyčka. Jedná se tedy o vzájemné zablokování procesů díky chybě synchronizačních primitiv. Nejjednodušší řešení je buď násilné ukončení procesů uvíznutých v deadlocku, popřípadě zrušení transakce a návrat k situaci těsně před ní (rollback)

2.3 Princip multitaskingu

V předchozí části této práce jsem uvedl a vysvětlil několik pojmů, které se úzce dotýkají tématu multitaskingu. Pokusil bych se tedy nyní shrnout to, jak multitasking probíhá a zároveň tak připravit prostor pro multithreading.

Jak tedy již bylo řečeno, multitasking znamená způsob, jak v jeden okamžik obsluhovat více úloh a procesů. Vzhledem k architektuře jednoprocesorových a jednojádrových systémů není samozřejmě možné, aby se tomu tak dělo opravdu v jeden okamžik. Je nutné, aby na těchto systémech existovala nějaká režie a správa střídání procesů na procesoru. Vzhledem k rychlým operacím a velmi krátkým časům, které jsou přidělovány procesům, je však z pohledu člověka možné považovat jejich běh za souběžný. Proto můžeme na počítači ve stejný okamžik poslouchat hudbu, psát dokument a mít otevřené další programy které potřebujeme. Právě to nám umožňuje multitasking. Ten je však až záležitostí moderních operačních systémů typu UNIX či Windows. Typickým příkladem jednoúlohového operačního systému je Microsoft DOS.

Multitasking dělíme na **skutečný** a **zdánlivý.** Skutečný multitasking je situace, kdy hardware počítače umožňuje, aby alespoň některé úlohy běžely opravdu souběžně. Je však nutné dodat, že ani u vícejádrových a víceprocesorových systémů v podstatě nikdy ke stoprocentnímu multitaskingu

nedojde a vždy je aspoň z části využíván multitasking zdánlivý, neboli pseudoparalelismus úloh.

U zdánlivého multitaskingu dále rozlišujeme multitasking kooperativní a preemptivní.

2.3.1 Kooperativní multitasking

Kooperativní multitasking je založen na aktivní spolupráci běžících úloh. Každý běžící proces po nějaké době musí předat řízení zpět operačnímu systému, odkud jej převezme jiná úloha. Ta po nějaké době opět odevzdá prostředky jiné. Z hlediska stability systému se nejedná o příliš bezpečné řešení – chybně naprogramovaná úloha způsobí zastavení a pád systému. Výhodou je snadná implementace na úrovni operačního systému.

2.3.2 Preemptivní multitasking

O odevzdávání a odebírání procesoru se stará pouze operační systém. V intervalu řádově milisekund přeruší běžící program, dojde k vyhodnocení situace – např. které úlohy žádají o přidělení procesoru, priority úloh, aktuální stav, atd. a podle toho se rozhodne, zda nechá běžet aktuální úlohu nebo odevzdá systémové prostředky jinému procesu. Samozřejmě je však nutné i implementovat dobrovolné přepnutí kontextu, úloha nemusí čekat celou dobu, kterou ji procesor přidělil. Může se vzdát prostředků např. pokud čeká na nějakou vstupně-výstupní operaci.

2.3.3 Implementace multitaskingu

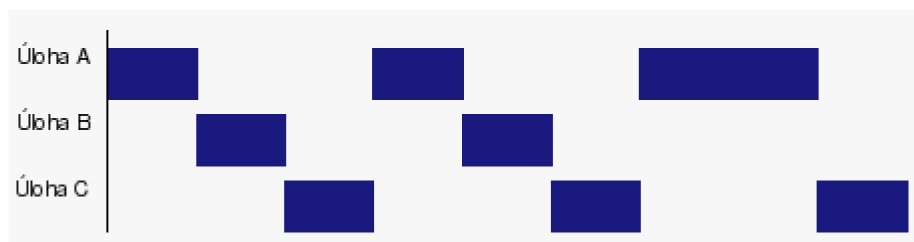
V první řadě je nutné říci, že existují operace u nichž musí být zaručeno, že proběhnou celé a jejich provádění nebude přerušeno. Takové operace se označují jako atomické. Každá atomická operace musí proběhnout buď celá, nebo vůbec. Příkladem atomické operace je např. použití synchronizačního

primitiva. Mezi zjištěním jeho stavu a následnou změnou nesmí být úloha přerušena, protože úloha, která by pak dostala řízení, by mohla stav primitiva změnit a došlo by k chybnému chování.

Ze složitějších (z hlediska implementace) technik meziprocesové komunikace zmíníme techniky **vysílací** a **přijímací**. U vysílacích technik dochází k tomu, že proces reaguje na zprávu obdrženou od operačního systému. U přijímacích technik je proces tou aktivní složkou a reaguje na základě monitoringu ostatních procesů a hodnot globálních proměnných. Nejjednodušší technikou je použití prostých smyček a volat je v cyklu.

Na úrovni operačního systému tedy existuje pole (většinou dynamické), obsahující seznam spuštěných procesů a ukazatelů na ně. Dále je nutná implementace fronty zpráv, které budou předávány procesu například jako parametr funkce. A operační systém musí mít definovány funkce pro přidávání a odstraňování procesů ze seznamu i zpráv z fronty, funkce pro rozesílání zpráv a samozřejmě jádro multitaskingového řešení, které obsahuje pole pro uložení procesů a funkce pro práci s nimi. Nad jádrem běží nekonečná smyčka, která obsluhuje procesy a zároveň jim rozesílá zprávy.

Celé schéma běžících úloh A, B, C pak na pseudoparalelním systému může vypadat následujícím způsobem.



Obr. 2 – Příklad rozdělování úloh v pseudoparalelním systému

2.4 Vlákno

Vlákno je speciální entita, kterou označujeme jako logickou část běžící aplikace. Platí, že každá spuštěná aplikace má na úrovni operačního systému alespoň jeden proces a každý proces má minimálně jedno vlákno. Můžeme se setkat i s označením podproces, já však v této práci budu označovat podproces jako vlákno, neboť z hlediska programovacího jazyka Java je takovéto označení mnohem přesnější. V analogii s předchozí teorií multitaskingu a procesů lze najít jisté styčné body mezi režií procesů a vláken, hierarchicky se můžeme dívat na vlákna jako na více procesů spuštěných v jednom procesu. Na rozdíl od běhu procesů však může programátor ovlivňovat běh vláken na úrovni vyššího programovacího jazyka.

U multithreadových aplikací tedy platí, že v rámci jednoho procesu může běžet několik vláken. Současně to znamená, že běží i v jednom paměťovém prostoru alokovaném operačním systémem pro proces se zdroji, které byly rodičovskému procesu přiděleny operačním systémem.

Výhodou je, že přepínání kontextu mezi vlákny je mnohem rychlejší, než přepnutí mezi procesy a je možné jej provést i bez účasti operačního systému, pouze na úrovni jazyka. Programátor tak může více zasahovat do běhu programu a přizpůsobit a optimalizovat jej přímo pro daný problém. S tím je však spojena i jistá nevýhoda. Vzhledem k tomu, že programátor má více možností jak ovlivnit vícevláknový běh své aplikace, musí se zároveň i postarat o konzistenci dat se kterými program pracuje. Znamená to vyřešit synchronizaci dat, běhu procesu, zámky sdílených dat.

Vícevláknové aplikace jsou mocným prostředkem k urychlení aplikace a zároveň jejímu optimálnímu běhu, ovšem v případě špatně napsaného kódu je chování programu nevyzpytatelné a nelze říci, jak se bude program chovat.

3 Úvod do paralelního programování v jazyce Java

V předchozí části této práce jsem konstatoval, že současné operační systémy jsou víceúlohové a umožňují v jeden okamžik spravovat několik úloh. Shodně byla tato myšlenka přenesena na úroveň programovacích jazyků jako **vlákna**. Stejně tak jako máme najednou spuštěných více programů v počítači, tak je toto možné i v rámci jedné aplikace. Například od textového procesoru očekáváme, že bude vždy připraven na vstup z klávesnice bez ohledu na to, zda právě formátuje text, zobrazuje menu či aktualizuje zobrazení. Software, který je takto navržený, označujeme jako vícevláknové aplikace, paralelní aplikace, popřípadě software s podporou souběžnosti.

Technika souběžnosti (**concurrency**) nám tedy umožňuje psát aplikace které vyhovují tomu, co jsem zmínil před chvílí. Platforma Java je od základů navržena tak, aby umožňovala programování paralelních aplikací. Podpora pro vláknové programování je k dispozici už v základních knihovnách jazyka a od verze Java 5 zahrnuje i víceúrovňová rozhraní API v knihovně `java.util.concurrent`.

S příchodem Javy 1.5 (Java 5) došlo k zásadnímu přepracování celého modelu paralelního programování v Javě. Java 5 nabízí kompletně předělaný a upravený paměťový model a spoustu nových tříd a rozhraní, které významným způsobem usnadňují práci s paralelními programy, zvyšují efektivitu programu a zároveň snižují těžkopádnost a náročnost kódu.

Přestože nové objekty a nový model paralelního programování přináší velké zjednodušení oproti minulým verzím, stále je třeba myslet na to, že správné řízení a běh programu je plně na programátorovi, který musí znát specifika javovské platformy a nesprávný běh programu a chybné výsledky výpočtu jsou jen a jen vinou špatné implementace a řešení problému. Přesto

však musíme říci, že od verze Javy 5 je psaní paralelních programů mnohem jednodušší.

Java 5 přináší přepracovaný paměťový model jazyka, který opravuje chyby předchozích verzí a přichází s komplexním řešením pro vícevláknové aplikace. Stejně tak knihovna `java.util.concurrent` nabízí objekty a rozhraní pro snadnější implementaci vícevláknovosti do aplikací.

3.1 Krátká historie technologie Java

Technologie Java vznikla jako nástroj pro programování aplikací v roce 1991 ve firmě Sun Microsystems. Na počátku byla snaha malého týmu zhruba dvaceti lidí o vývoj jazyka a platformy, která by znamenala „novou vlnu“ ve tvorbě aplikací. Zhruba po osmnácti měsících snažení vznikl první výsledek, tehdy ještě pod kódovým jménem Oak. Cílem bylo vytvořit platformu, která bude nezávislá na hardwaru, na kterém poběží a díky jednomu jazyku bude možné psát aplikace nezávisle na systému, na kterém bude aplikace nasazena.

Ze začátku se tento nápad nesetkal s příliš velkým zájmem, to se však změnilo v polovině devadesátých let spolu se začínajícím boomem internetu a spoluprácí s firmou Netscape a jejich webovým prohlížečem Navigator. Od roku 1996, kdy byla představena první verze Javy, došlo k jejímu obrovskému rozmachu.

V současné době je Java využívána několika miliony softwarovými vývojáři jako jejich hlavní pracovní nástroj a je nasazena údajně na více než pěti miliardách zařízení (PC, mobilní zařízení, „smart cards“ – speciální verze Javy přizpůsobená pro běh na různých zabezpečovacích kartách a čipech, dále je součástí např. set top boxů, tiskáren, navigačních systémů či různých automatů).

3.2 Filosofie Javy

Java je technologie pro vývoj a provoz počítačových aplikací. Její součástí je programovací jazyk Java a počítačová platforma shrnující různé varianty použití programovacího jazyka Java pro vývoj a provoz různých typů aplikací. Ta je rozdělena na následující dílčí platformy:

- **JavaCard**
  Je určena pro aplikace provozované na takzvaných chytrých kartách (smart cards), např. platební a kreditní karty
- **Java ME (Micro Edition)**
  Je vyvinuta pro provoz na přenosných zařízeních (mobilní telefony, PDA)
- **Java SE (Standard Edition)**
  Pro aplikace určené pro klasické stolní počítače a notebooky
- **Java EE (Enterprise Edition)**
  Aplikace pro podnikové a rozsáhlé informační systémy

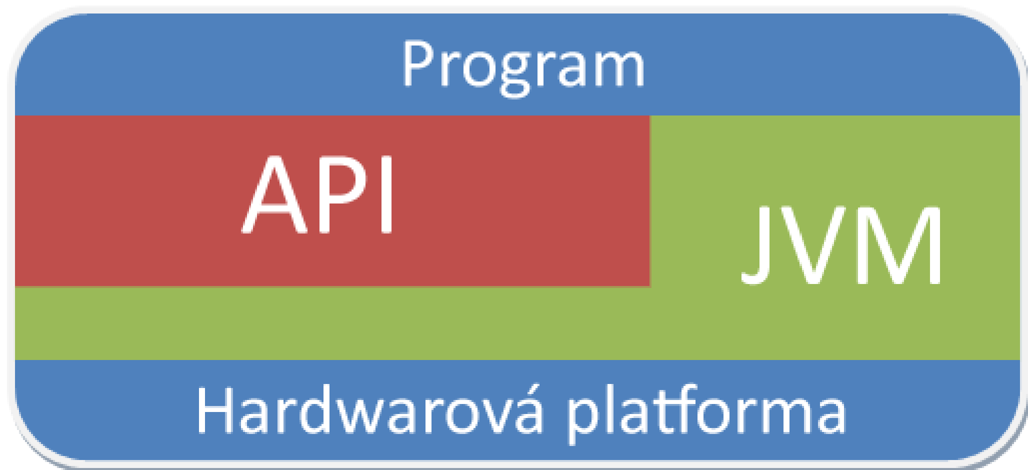
V této práci se budeme zabývat vláknovým a paralelním programováním na Java SE a to na Javě ve verzi 6.

3.2.1 Platforma Java

Platforma je prostředí, které umožňuje spouštět aplikace. Většinou pokud mluvíme o platformě, rozumíme tím zpravidla kombinaci hardwaru a softwaru. Nejrozšířenější kombinací je platforma typu operační systém Microsoft Windows nad architekturou PC, dále např. systémy UNIXového typu nad stejnou architekturou. Java jako platforma se od ostatních liší tím, že se jedná o čistě softwarovou platformu fungující nad jinými hardwarovými platformami, v podstatě ji můžeme popsat jako mezičlánek či nadstavbu nad operačním systémem.

Platforma Java zahrnuje dvě komponenty

- Virtuální stroj jazyka Java (JVM)
- Aplikační programové rozhraní (API)



Obr. 3 – Izolace programu od hardwaru

3.2.1.1 Java virtual machine (JVM)

Z filosofie Javy, jakožto jazyka nezávislého na architektuře vyplývá, že program nemůže být překládán no nativního kódu pro určitou platformu, nýbrž je nutné nějak zajistit jeho snadnou přenositelnost. O to se stará právě JVM. Zdrojový kód javovského programu je uložen v textovém souboru s příponou .java, pomocí kompilátoru *javac* je přeložen do tzv. bytového kódu (bytecode), který je uložen v souboru s příponou .class. V souboru .class není nativní kód optimalizovaný pro příslušný procesor, nýbrž strojový kód pro Java virtual machine. Ten je pak spouštěn nad javovskou platformou pomocí nástroje *java* a nezávisle na tom, na jakém systému je spouštěn.

Obr. 4 – Schéma běhu programu v Javě

Portace JVM je k dispozici pro mnoho operačních systémů. Proto lze (samozřejmě nad základní instalací Javy bez různých přidaných knihoven a modulů) stejný soubor .class spustit v systémech Windows stejně tak jako Solaris, Linux nebo Mac OS.

Protože Java představuje prostředí nezávislé na hardwarové platformě, mohou být někdy programy pomalejší než nativní kód, což je díky mezičlánku v podobě JVM pochopitelné. V posledních verzích Javy je však při běhu již téměř neznatelný rozdíl oproti nativním aplikacím v jiném jazyce. Deficit je však pořád ještě znát při startu aplikace, neboť se program nejdříve překládá a až pak spouští. Je však možné využít mechanismů JIT (Just in time compilation) a HotSpot, kdy se často prováděné nebo neefektivní části kódu přeloží do strojového kódu a program se zrychlí, popřípadě se málo používané části překládají až ve chvíli, kdy jsou potřeba.

3.2.1.2 Java API

Java API představuje rozsáhlou sbírku hotových komponent, které poskytují mnoho možností. Člení se do knihoven se souvisejícími třídami a rozhraními, v terminologii Javy se tyto knihovny označují jako balíčky (packages). Základní třídy najdeme v balíčku `java.lang`, třídy pro práci s datovými proudy v `java.io` atd.

3.3 Programovací jazyk Java

Java je čistě objektově navržený programovací jazyk vyšší úrovně. Mezi jeho klíčové vlastnosti patří:

- **Jednoduchost syntaxe**
  Java vychází z jazyků typu C, je uzpůsobena co nejsnadnějšímu zápisu kódu
- **Nezávislost na architektuře, přenositelnost, interpretovanost**
  Díky JVM uzpůsobující bytekód pro konkrétní platformu na které je spouštěn
- **Robustní a bezpečný jazyk**
  Správa paměti je čistě záležitostí JVM, tzv. Garbage Collector automaticky uvolňuje paměť, kterou zabírají objekty, se kterými se již nebude pracovat. Programátor také v podstatě nemá žádné prostředky, jak ovlivňovat práci s pamětí, čímž se vyhýbá řadě možných chyb. Zároveň je Java uzpůsobena pro nasazení v síťovém prostředí a chrání počítač před nebezpečnými operacemi
- **Výkonný a víceúlohový systém**
  Java je navržena přímo pro podporu multithreadových aplikací, zároveň díky množství optimalizací jak při překladu tak při běhu je velmi výkonná
- **Dynamičnost**
  Java byla navržena pro nasazení v rychle se vyvíjejícím a měnícím prostředí. Knihovna může být dynamicky za chodu rozšiřována o nové třídy a funkce, a to jak z externích zdrojů, tak programem samotným

3.4 Java Memory Model

K tomu, aby bylo možné pochopit, jak pracovat s vlákny, je nutné vědět něco o technickém pozadí celého procesu běhu programu a vlákna. Jak bylo řečeno v předchozí části, Java jako taková je izolována od operačního systému a hardwaru a ona je tím prostředím, na němž naše aplikace běží. Java jako softwarová nadstavba tak musí mít na starosti i práci s pamětí, javovské programy nemají paměť přidělovánu od operačního systému, ale zprostředkovaně právě přes JVM. Tento proces má na starosti tzv. Java Memory Model (JMM).

Pro rozsah a potřeby této práce potřebujeme vědět, že Java Memory Model definuje vztah mezi proměnnými našeho programu a tím, jak jsou uloženy a získávány z paměti počítače. V předchozích verzích Javy bohužel nebyl tento model úplně dokonalý. Byl vhodným základem pro tvorbu vícevláknových aplikací, nebyl však moc intuitivní a navíc vzhledem k různým optimalizacím na úrovni práce s pamětí nebyl ani bezpečný ve vztahu k datům ke kterým vlákna přistupovala. Proto došlo k revizi paměťového modelu, a výsledkem byla formulace JSR 133, kde došlo k novému návrhu a zároveň s tím vznikly i nové konstrukce a nová filosofie pro paralelní programování v Javě. Všechny tyto záležitosti se objevily v Javě 5.

Existují dva typy memory modelu, které se liší v sémantice práce s pamětí:

- **Strong memory model**
  Je intuitivní a přirozený pro práci jednovláknové aplikace. Všechny zápisy do paměti jsou viditelné v takovém pořadí, v jakém proběhly. Přerovnání instrukcí kompilátorem sice nemají vliv na vlastní sémantiku programu, pro vícevláknové aplikace je ale nevýhodný protože výrazně omezuje optimalizační možnosti procesoru nebo interpretu jazyka a tím i výkon.

- **Weak memory model**
  Tento typ využívá i Java, byl revidován právě ve verzi Java 5. Tento model

se pro jednovláknové aplikace chová shodně jako strong, situace je ale složitější u vícevláknových programů. Může se například stát, že zápis do dvou proměnných jedním threadem uvidí druhý thread v jiném pořadí nebo dokonce vůbec. Program pak může vracet různé výsledky pro každé spuštění, optimalizace kompilátoru jsou na úrovni jazyka neovlivnitelné, a nelze určit, v jakém pořadí budou spuštěná vlákna obsloužena procesorem. Proto Java nabízí nástroje pro zajištění konzistence programu a dat – bariéry, kritické sekce, atomické instrukce.

Při práci s vlákny je tedy nutné brát na vědomí hlavně dvě věci – ošetřit viditelnost proměnných mezi jednotlivými vlákny a to, že může dojít k přeskupení pořadí zápisu a čtení proměnných. Při práci s proměnnou může totiž docházet k optimalizacím jak na úrovni Java Virtual Machine (optimalizace ve smyslu JIT nebo HotSpot) i na úrovni procesoru, proměnná může být uložena v cache nebo registru CPU a vlákna tak mohou pracovat s nekonzistentními či neaktuálními daty.

Na Java Memory Model se můžeme dívat i jako na abstrakci v popisování toho, jaké optimalizace proběhnou nad procesorem, cache a registry. Java Memory Model pracuje s proměnnými na dvou úrovních paměti – **hlavní paměť**, která je sdílena všemi vlákny, a **lokální paměť**, která je naopak privátní pouze pro konkrétní vlákno. V případě, že vlákno změní hodnotu v proměnné, promítne se tato změna do lokální paměti. Měla by být samozřejmě reflektována i v hlavní paměti, ale díky absenci jakékoliv synchronizace nemůžeme určit, v jakém čase k tomu dojde. Stejně tak v případě změny hodnoty proměnné v hlavní paměti může chvíli trvat, než se změní i hodnoty v lokální paměti a všechna vlákna tak mohou získat aktuální hodnotu.

Tento model počítá s optimalizacemi na úrovni procesoru, s prodlevami během přesouvání dat mezi různými typy paměti, či s reorganizací a přeskupováním příkazů procesorem a kompilátorem jazyka. Ve většině případů tak nedojde k žádným změnám, bohužel v případě vícevláknových

aplikací se mohou objevit problémy. Java Memory Model sám rozhoduje o tom, kdy je vhodné provést optimalizaci a vzhledem k různým
časům, algoritmům přeskupování a optimalizačním postupům se může stát, že například jedno vlákno zapisuje dvě proměnné v určitém pořadí, a druhé vlákno je čte v opačném.

Pokud tedy víme, že bude docházet k situaci, kdy jedno vlákno zapisuje hodnoty a krátce po něm druhé vlákno tyto hodnoty čte, musíme tuto situaci vhodně ošetřit. Nevíme totiž, jaký výsledek můžeme dostat a proto musíme tyto události vhodně programově synchronizovat.

```
class SynchronizationProblem
{
   int x = 0;
   int y = 0;

public void writer()
{
    x = 1;
    y = 2;
}

public void reader() {
    int r1 = y;
    int r2 = x;
  }
}
```

Vlákno A spustí metodu `writer()` a vlákno B metodu `reader()`. Předpoklad, že po proběhnutí tohoto příkladu bude v proměnné `r1` hodnota 2 a v `r1` hodnota 1 **při absenci synchronizace** však vůbec nemusí být pravda. Může dojít jak k přehození, tak k tomu, že vlákno B pracuje s hodnotami 0, které jsou uloženy v cache.

4 Základy práce s vlákny v jazyce Java

V předchozích kapitolách bylo vysvětleno, jak operační systém pracuje s procesy, jaké je technologické pozadí víceprocesových aplikací a v jaké míře lze toto všechno aplikovat na platformu Java. Bylo také zmíněno, že vlákna jsou abstrakcí procesů a lze na ně v podstatě aplikovat převážnou většinu toho, co jsme řekli o procesech a práci s nimi. Vlákna v jazyce Java se však liší v tom, že o práci s pamětí a daty, ke kterým přistupují, se nestará operační systém, ale Java Virtual Machine a její část Java Memory Model. V případě několika spuštěných aplikací nás nemusí zajímat, jakým způsobem dochází k jejich přepínání, jak přistupují k datům a jak je zajištěna jejich vzájemná jedinečnost a bezpečnost. O tyto věci se stará operační systém, a vše je implementováno na nejnižší úrovni tak, aby uživatel nemusel o ničem vědět.

Ve chvíli, kdy vyvíjíme vícevláknovou aplikaci však režii přebírá Java Virtual Machine a ošetření konzistence dat je úkolem programátora. Java Virtual Machine a Java Memory Model poskytnou technologické zázemí, ale není v jejich silách odhadnout, jak má námi navrhovaná aplikace zacházet s daty, jakým způsobem chceme sdílet hodnoty mezi jednotlivými vlákny a jak chceme řídit jejich běh a interakci mezi sebou. Optimalizace nad kompilátorem a CPU se chovají jako dvojsečná zbraň, významným způsobem zvyšují výkon spuštěné aplikace, ale také mohou vést právě k nekonzistenci dat a vracení nerelevantních výsledků, pokud programátor tyto situace neodchytí a neurčí, jak se v nich má aplikace chovat a jakým způsobem data budou zpracovávána.

Správně napsaná paralelní aplikace bude mít vysoký výkon i na jednojádrových jednoprocesorových systémech díky tomu, že Java byla navrhnuta jako vícevláknový jazyk. Na platformách s více jádry a procesory tento výkon ještě velmi rychle poroste.

Java 5 nabízí velmi mocné prostředky pro práci s vlákny a urychluje návrh vícevláknových aplikací napsáním minimálního objemu kódu, nabízí rozhraní a třídy, které s vlákny dokáží pracovat za využití velmi malého úsilí. Součástí API jsou i třídy pro atomickou práci s daty, implementace synchronizačních primitiv, zámky, speciální kolekce pro práci s paralelními programy a další objekty navržené pro zajištění konzistence dat a atomicity práce s nimi. Do jisté míry můžeme ovlivnit i přerovnávání přístupu k paměti pomocí bloků kódu, do kterých nesmí zasáhnout přerovnávání a je u nich zajištěno správné a kompletní zapsání do paměti.

4.1 Objekty typu Thread

Objekty typu `Thread` jsou v jazyce Java abstrakcí pro podprocesy a základními objekty, se kterými budeme pracovat při navrhování paralelních programů. Budu je tedy v této práci označovat jako **vlákna**, tento název nejlépe vyhovuje terminologii jazyka Java. Jak bylo řečeno již dříve, vlákna existují v rámci procesu. Každý spuštěný proces obsahuje alespoň jedno vlákno, které zároveň sdílí všechny prostředky procesu, včetně paměti nebo otevřených souborů. Vytvoření vlákna si vyžádá i menší systémovou režii než vytvoření procesu, a díky sdílení prostředků procesu nabízí účinné, ale zároveň i lehce problematické možnosti komunikace. Každý program na začátku obsahuje jedno vlákno, označované jako hlavní vlákno (Main Thread), které může dále vytvářet další vlákna.

Každé vlákno je přidružené k instanci třídy `Thread` z balíčku `java.lang`. Java rozeznává dva typy vláken – klasické vlákno a vlákno typu **daemon**. Daemon znamená vlákno, které běží na pozadí, kde vykonává průběžně nějakou činnost a není za normálních okolností ukončováno. Obvykle pracuje jako rezidentní vlákno, bez jakékoliv interakce.

4.1.1 *Životní cyklus vlákna*

V každém okamžiku se vlákno nachází právě v jednom z následujících stavů:

- **Nové vlákno *(New)***

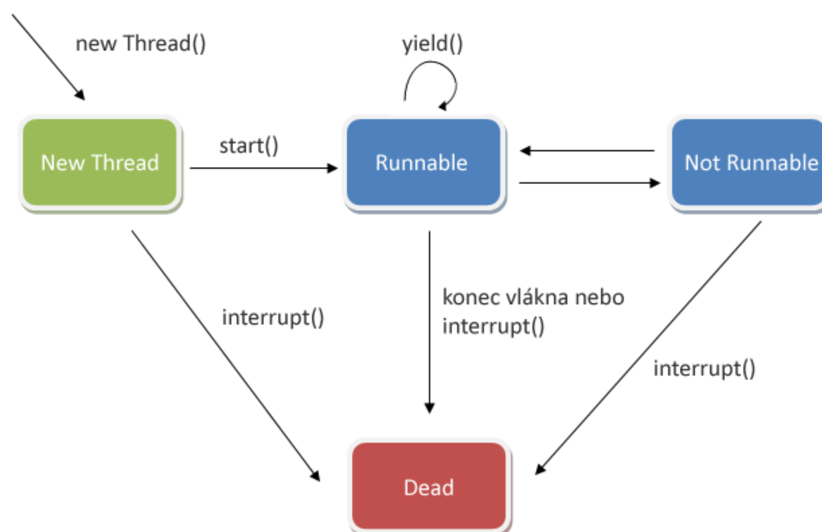
  Vlákno je vytvořeno, ale ne inicializováno – čeká na spuštění
- **Běžící vlákno *(Runnable)***

  Vlákno je spuštěné a provádí nějakou činnost
- **Pozastavené vlákno *(Not Runnable)***

  Vlákno je spuštěné, ale v tuto chvíli přerušené a neprovádí žádné instrukce
- **Ukončené *(Dead)***

  Vlákno ukončilo svou činnost, vše co s ním bylo spojené odstraní Garbage Collector

Mezi jednotlivými stavy přechází vlákno téměř vždy pomocí volání metod třídy `java.lang.Thread` tak, jak je ukázáno na následujícím schématu.



Obr. 5 – Schéma běhu vlákna

4.1.2 Priority vlákna

Nastavení toho, jak často bude kterému vláknu přidělen procesor, a systémové prostředky je plně v režii Java Memory Modelu. Java používá **metodu plánování podle priority** (kapitola 2.2.7) Pro toto nastavení platí, že každé vzniklé vlákno má nastavenu prioritu podle rodičovského vlákna, ve výchozím případě, pokud není určeno jinak, je tato priorita nastavena na normální. Java definuje stupnici priorit vláken od 1 do 10, přičemž 1 znamená nejnižší prioritu a 10 nejvyšší. Součástí třídy `Thread` jsou také tři konstanty definující nejběžnější hodnoty:

`static int MIN_PRIORITY` pro hodnotu 1,

`static int NORM_PRIORITY` pro hodnotu 5

`static int MAX_PRIORITY` pro hodnotu 10

Zde však chci podotknout, že nevidím smysl v používání priorit vláken, protože tím přicházíme o hlavní výhodu multivláknových aplikací. Ve chvíli, kdy nastavíme vláknům priority, tak dochází vlastně k sekvenčnímu spouštění vláken a kód je tak vykonáván stejně, jako by nebyl naprogramován paralelně. Z definice algoritmu plánování pomocí priorit totiž jasně plyne, že všechna vlákna jsou spouštěna podle priorit od nejvyšší po nejnižší a vlákno s nižší prioritou nemůže převzít řízení, pokud není ještě ukončeno vlákno s prioritou vyšší.

Nejlépe to demonstruje následující příklad *(projekt 412_-_priorita-vlaken)*:

Ukázka kódu:

```
public static void main(String[] args)
{
      Vlakno v1 = new Vlakno("Vlakno 1: ");
      Vlakno v2 = new Vlakno("Vlakno 2: ");
      v2.setPriority(Thread.MAX_PRIORITY);
      v1.start();
      v2.start();
}
```

V prvním případě necháme program proběhnout tak, jak je. Ve druhém odstraníme příkaz `v2.setPriority(Thread.MAX_PRIORITY);` a bude nás zajímat výpis do konzolového okna.

s prioritou	bez priority
Vlakno 2: 0 Vlakno 2: 1 Vlakno 2: 2 Vlakno 2: 3 Vlakno 2: 4 Vlakno 1: 0 Vlakno 1: 1 Vlakno 1: 2 Vlakno 1: 3 Vlakno 1: 4	Vlakno 1: 0 Vlakno 2: 0 Vlakno 2: 1 Vlakno 1: 1 Vlakno 2: 2 Vlakno 1: 2 Vlakno 2: 3 Vlakno 1: 3 Vlakno 2: 4 Vlakno 1: 4

Vidíme tedy, že vlákno s normální prioritou muselo počkat na to, až bude ukončeno vlákno s námi nastavenou vysokou prioritou. V případě, že obě vlákna měla nastavenu prioritu na stejné úrovni, docházelo k jejich zhruba pravidelnému střídání. Výsledek však můžeme dostat vzhledem k chybějící synchronizaci vláken jiný pro každé spuštění programu. Není totiž zaručeno, že se vlákna budou opravdu pravidelně střídat.

4.2 Vytvoření vlákna

Jak již bylo řečeno, každé vlákno je přidruženo k instanci třídy `java.lang.Thread`. Existují dva způsoby, jak vytvořit instanci třídy `Thread`.

- Navrhneme třídu, která implementuje rozhraní `java.lang.Runnable` a její instanci předáme jako parametr konstruktoru třídy `java.lang.Thread`
- Třídu odvodíme jako potomka třídy `java.lang.Thread`

4.2.1 Vytvoření vlákna pomocí rozhraní Runnable

Tento postup je obecnější a jeho výhodou je, že můžeme vytvořit potomka jiné třídy, než je `Thread`. Rozhraní `Runnable` definuje jedinou metodu `run()`, která je určena k umístění kódu, který bude vlákno vykonávat. Naše třída implementuje rozhraní `Runnable` a její instance potom předáme jako parametr konstruktoru třídy `Thread`.

(Projekt 421_-_vytvoreni-vlakna-pomoci-runnable)

```
public class VlaknaRunnable implements Runnable
{

    public void run()
    {
        System.out.println("Vytvoreni vlakna pomoci
Runnable");
    }

    public static void main(String[] args)
    {
        (new Thread(new VlaknaRunnable())).start();
    }
}
```

4.2.2 Vytvoření vlákna odděděním od třídy Thread

Tento případ je snadnější a trochu přehlednější, jsme však omezeni tím, že naše třída musí být potomkem `java.lang.Thread`. Sama třída `java.lang.Thread` implementuje rozhraní `Runnable`, nemá však

překrytu metodu `run()`. To nám umožní vytvořit vlastní implementaci metody `run()`.

(Projekt 422_-_vytvoreni-vlakna-pomoci-dedicnosti)

```
public class VlaknaThread extends Thread
{

    public void run()
    {
        System.out.println("Vytvoreni vlakna dedenim");
    }

    public static void main(String[] args)
    {
        (new VlaknaThread()).start();
    }
}
```

Výhodou prvního přístupu je i to, že z objektového hlediska odděluje úkol, který vykonává `run()` od objektu `Thread`. Získáváme tak nejen pružnější návrh, ale i možnost aplikovat jej na vysokoúrovňová rozhraní API pro správu vláken.

4.3 Pozastavení běhu vlákna

Vlákno lze nejen spustit, ale můžeme i ovládat jeho běh. První možností je pozastavení jeho činnosti. Pomocí tohoto postupu můžeme cíleně pozastavit běh vlákna a zpřístupnit tak procesor jiným vláknům. Lze tímto způsobem také ovlivnit tempo činnosti programu. Je však nutné podotknout, že tato funkce záleží na operačním systému a není zaručeno, že pozastavíme proces na přesně určenou dobu.

Tuto událost provede metoda `static void Thread.sleep(long millis)` z třídy `Thread`, případně její přetížená verze `sleep(long millis, int nanos)`. Parametry znamenají dobu v milisekundách a nanosekundách, na kterou se běh vlákna pozastaví. Obě metody vyhazují

výjimku `java.lang.InterruptedException`, kterou je nutné buď zachytit nebo deklarovat vyhození.

Následující příklad ukazuje vytvoření jednoduchého čítače, který bude ve vteřinových intervalech vypisovat čísla od jedné do deseti. Zároveň ukážeme, že i metoda `main()` je vlastně vláknem, neboť ji lze pozastavit pomocí metody `sleep()`. Tím demonstrujeme tvrzení, že každý program má alespoň jedno vlákno.

(Projekt 430_-_pozastaveni_vlakna)

```
public static void main(String[] args)
    {
        for (int i = 1; i <= 10; i++)
        {
            try
            {
                System.out.println(i);
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                System.out.println("Doslo k preruseni
programu");
            }
        }

        System.out.println("Konec behu programu");
    }
```

Třída `Thread` nabízí ještě jednu metodu pro ovlivnění běhu vlákna, kterou můžeme považovat do jisté míry za pozastavení. Je jí metoda `yield()`, která způsobí, že právě běžící je pozastaveno a vzdá se systémových prostředků ve prospěch jiných vláken. Pozastavení trvá pouze do doby, než na toto vlákno přijde řada při dalším cyklu přidělování procesoru. Můžeme tedy říci, že vlákno se po zavolání metody `yield()` dobrovolně vzdá prostředků, pokud neexistuje žádná smysluplná činnost, kterou by mohlo vykonávat a zbytečně tak snižovat výkon.

4.4 Přerušení běhu vlákna

Přerušení (interruption) představuje zprávu pro vlákno, že by mělo zastavit právě prováděnou činnost. Tato zpráva může být vyvolána jak operačním systémem, tak jiným vláknem pomocí metody `java.lang.Thread.interrupt().` Tato metoda nastaví tzv. příznak přerušení a tím říká ostatním, že vlákno je přerušeno.

S událostí přerušení je spjata výjimka `InterruptedException`, při jejím zachycení pak můžeme celou událost ošetřit. Metody `sleep()`, `join()` a `interrupt()` ze třídy `Thread` tuto výjimku vyhazují a předávají řízení. Programátor se tedy musí o tuto událost postarat ve svých metodách v případě, že je možné přerušení vlákna očekávat. Nejjednodušším řešením je volání metody boolean `Thread.interrupted()`, která vrací true v případě, že je vlákno přerušeno a dále událost ošetřit, elegantním řešením je například programové vyhození výjimky `InterruptedException`.

```
if(Thread.interrupted())
   throw new InterruptedException();
```

4.4.1 Shrnutí – projekt Práce s vlákny

V tomto příkladu ukážeme jednoduchou práci s vlákny. Máme zde dvě vlákna. **main** (které je součástí každého programu) vytvoří objekt typu **Vlakno**, se kterým pak dále pracuje. Pokud vlákno Vlakno trvá příliš dlouho, je ukončeno. Vlakno vypisuje sérii zpráv, načítaných z pole. Pokud dojde k přerušení před vypsáním všech zpráv, bude vypsána zpráva o chybě, a vlákno bude ukončeno.

(Projekt 44_-_zaklady-prace-s-vlakny)

4.5 Úvod do synchronizace vláken

Vlákna zpravidla komunikují tak, že sdílejí společně přístup k datovým složkám a datům. Tento způsob komunikace a její řešení je sice mimořádně efektivní, ale bohužel vede i k chybám. První chybou je tzv. **interference vláken** (více vláken přistupuje v jeden okamžik ke stejným datům), tou druhou pak **chyba v konzistenci paměti** (více vláken má nekonzistentní zobrazení dat, která by však měla být stejná). To, že běh vláken není navzájem ošetřen má za důsledek na jednu stranu rychlejší paralelní běh, problém však nastane ve chvíli, kdy vlákna sdílejí data. Pro programátora tak není podstatný vznik těchto chyb, ale strategie, jak se jim vyhnout.

4.5.1 Interference vláken

K interferenci vláken může dojít v případě, že dvě vlákna pracují se stejnými daty. Obě operace zahrnují delší sekvence a navzájem se tak překrývají. Uveďme příklad:

(projekt 451_-_interference-vlaken)

```
class Counter
{
   private int c = 0;

   public void zvysit()
   {
         c++;
   }

   public void snizit()
   {
         c--;
   }

   public int vratHodnotu()
   {
         return c;
   }
}
```

Na první pohled se může zdát, že nemůže dojít k tomu, aby k nějaké interferenci mezi procesy došlo. Java Memory model však funguje tak, že ani u těchto jednoduchých příkazů není zaručena jejich atomicita. Může tak dojít k situaci, že zatímco probíhá inkrementace, další proces zároveň provádí dekrementaci. Tento příkaz totiž není vykonáván v konstantním čase a není ani atomický – lze jej rozložit jako načtení hodnoty `c`, zvýšení načtené hodnoty o 1 a zápis do paměti.

Uveďme si jeden příklad, jak může situace vypadat. Mějme vlákna A a B, obě budou přistupovat k metodám třídy `Counter` a proměnné `c`. Snadno může dojít k tomu, že vlákna A i B načtou hodnotu `c`, ta je 0. Vlákno A provede inkrementaci, vlákno B dekrementaci. Proměnná c ve vláknu A tak má hodnotu 1, ve vláknu B je -1. A nyní se obě vlákna pokusí výsledek zapsat. Logicky dojde k tomu, že dostaneme nesprávný výsledek. Teoreticky bychom měli dostat výsledek 0, ale vzhledem k nepředvídatelnosti nesynchronizovaných vláken nemůžeme určitě říct, jaká hodnota bude v proměnné `c`. Právě díky tomu, že nevíme jak se interpret Javy zachová, a v jakém pořadí vlákna budou probíhat, je nutné použít nějakou techniku ošetření této situace.

4.5.2 Chyby v konzistenci paměti

K této chybě dojde v případě, že vlákna přistupují k datům, která by měla být shodná, ale nejsou. Příčinu hledejme v tom, jak Java Virtual Machine pracuje s pamětí a ve weak memory modelu. V případě vícevláknového programu nemáme nikdy zaručeno, jak dlouho bude trvat zápis hodnoty do paměti a zda se to stihne ještě před tím, než tuto hodnotu bude chtít přečíst jiné vlákno. V Javě totiž není zaručena atomicita pro primitivní typy ani u takové operace, jakou je zápis do paměti.

Provedeme modifikaci příkladu *451_-_interference-vláken* a přidáme metodu `vypisHodnotu()`.

```
public static void vypisHodnotu()
{
   System.out.println(c);
}
```

Vlákno A provede metodu `zvysit()` a vlákno B metodu `vypisHodnotu()`. Pokud by obě metody proběhly v rámci jediného vlákna, bude hodnota proměnné `c` rovna 1, tedy přesně taková, jakou očekáváme. Pokud však tuto úlohu zasadíme do vícevláknového prostředí, může se stát, že metoda `vypisHodnotu()` nám vypíše 0, protože Java Memory Model ještě nestihl zapsat novou hodnotu proměnné `c`.

Řešením tohoto problému je programové vytvoření vztahu nazývaného „**nastává-před**“ (happens-before). Tento vztah nám zaručí, že obsah paměti, zapsaný jedním příkazem, bude viditelný jinému příkazu tak, jak má být a nedojde k nekonzistenci paměti.

Vztah „nastává-před“ je vlastně relací mezi námi vykonávaným příkazem a příkazy, které mu následují. Zjednodušeně můžeme říci, že pokud A a B jsou příkazy v rámci vláken a jsou v relaci „nastává-před“, pak to znamená, že příkaz B nemůže být vykonáván pokud není ukončen příkaz A. V důsledku tak máme zaručenu konzistenci dat.

Některé metody v Javě mají tento vztah již implementován (např. vytvoření či zrušení vlákna), v našich třídách a programech se o to musíme postarat sami. Java nám nabízí několik možností, jak toho docílit. My se v této části zmíníme o těch nejdůležitějších pro tzv. nízkoúrovňovou synchronizaci vláken. Pro práci se synchronizací na vyšší úrovni nabízí prostředky knihovna `java.util.concurrent` a pojednává o ní kapitola 5.

4.6 Synchronizace

Nejjednodušší způsob, jak zajistit konzistenci a synchronizaci dat je použití klíčového slova `synchronized`. To má za následek dvě důležité věci:

- Není možné, aby došlo k interferenci vláken a dat, ke kterým přistupují. Pokud vlákno volá synchronizovanou metodu, dojde automaticky k pozastavení vláken, která pomocí **synchronizovaných** metod přistupují k datům, se kterými právě tato metoda pracuje.
- Po ukončení synchronizované metody dojde k vytvoření relace „nastává-před" mezi právě ukončenou metodou a všemi následujícími voláními **synchronizovaných** metod stejného objektu. Tím máme zaručeno, že jakákoliv změna objektu bude správně viditelná všem vláknům.

Důsledkem toho, že tento vztah platí jen pro synchronizovaná vlákna, je větší efektivita aplikace, protože programátor sám určí, kterých vláken se to týká. Ostatní vlákna mohou vykonávat jinou činnost a blokace se jich netýká.

Synchronizované metody jsou nejjednodušší strategií jak předcházet interferenci vláken a chybám konzistence paměti. Pokud je objekt viditelný pro více procesů, použitím klíčového slova `synchronized` pro všechny metody, které k němu přistupují, získáme účinný nástroj pro zajištění správnosti výsledků.

4.6.1 Vnitřní zámky a monitor, fungování synchronizace

Synchronizace v Javě je založena na vlastnosti, která se označuje jako **vnitřní zámek** (inner lock). Z hlediska synchronizačních primitiv se jedná o monitor, který zajišťuje správnou činnost synchronizace. Zajišťuje výhradní přístup k objektu i vytvoření relace „nastává-před". Každý objekt v Javě má se sebou svázán monitor. Při běhu programu je pak vlastnictví zámku výhradní záležitostí – objekt, který chce zámek získat o něj musí nejprve požádat současného vlastníka zámku, dojde k uvolnění zámku od současného

majitele, získá jej objekt, který žádal a nakonec musí zámek opět uvolnit. Pokud vlákno vlastní vnitřní zámek, nemůže jej získat žádné jiné vlákno, a při pokusu o získání zámku dojde k zablokování žádajícího procesu. Zároveň je ošetřeno vytváření relace „nastává-před“ při každém uvolnění zámku, čímž máme zaručenu konzistentnost dat mezi vlákny. Ve chvíli, kdy vlákno tedy zavolá synchronizovanou metodu, získá vnitřní zámek objektu, jemuž daná metoda patří. Zámek opět uvolní až po dokončení metody, ať už předáním řízení nebo vyhozením výjimky. [1]

Monitor má tedy tři funkce:

- **Exkluzivita přístupu**
  V jeden okamžik může vlastnit monitor pouze jedno vlákno
- **„Podmínka čekání“ (wait condition)**
  Jde v podstatě o to, že vlákno, které chce získat monitor, musí počkat dokud jej aktuální vlákno opět neodevzdá. Využívá se volání metody `wait()` a notifikace vláken pomocí `notifyAll()`
- **„ Paměťová zábrana“ (memory barrier)**
  Před blok synchronized se do bytekódu programu vkládá instrukce monitorenter, která slouží jako tzv. ***read-barrier*** – po této instrukci musí veškerá čtení z paměti získat aktuální hodnoty. Zároveň je na konec synchronized bloku vložena instrukce monitorexit, která se chová jako ***write-barrier***, což znamená, že veškeré hodnoty, zapsané v kritické sekci, budou po jejím ukončení viditelné ostatním vláknům.

4.6.2 Synchronizované metody

[1] V případě statických metod je vnitřní zámek přiřazen k objektu typu `Class`, který souvisí s danou třídou. Je tedy nezávislý na zámku instance třídy.

Provedeme jednoduchou modifikaci příkladu Counter, abychom získali bezpečný kód se synchronizovanými metodami.

(projekt 462_-_synchronizovany-citac)

```
class SynchronizedCounter
{
   private int c = 0;

   public synchronized void zvysit()
   {
         c++;
   }

   public synchronized void snizit()
   {
         c--;
   }

   public synchronized int vratHodnotu()
   {
         return c;
   }
}
```

Jednoduchým přidáním klíčového slova `synchronized` nyní máme v souladu s tím, co jsme uvedli na začátku kapitoly, zaručenu konzistenci dat při použití programu ve vícevláknovém prostředí.

4.6.3 Synchronizované příkazy

Jedná se o jiný způsob vytvoření synchronizovaného kódu. Musíme však určit objekt, který poskytuje vnitřní zámek. Výhoda synchronizovaných příkazů je v tom, že nemusíme psát celé synchronizované metody, ale zesynchronizovat pouze ty části kódu, které potřebujeme. Zároveň musíme určit objekt, který poskytne zámek. Toto řešení je výkonnější, ale zároveň náchylnější k chybám.

Využití si ukážeme na jednoduché metodě, která slouží pro paralelní vkládání do kolekce.

(projekt 463_-_synchronizovane-prikazy)

```
public void pridejZboziDoKolekce(String nazev)
{
   synchronized(this) //použití aktuální instance jako zámku
   {
         nazevVyrobku = ”Nazev vyrobku: ” + nazev;
         pocetVyrobku++;
   }
   seznamVyrobku.add(nazevVyrobku);
}
```

V tomto případě je vhodné vyhnout se volání metod jiného objektu (zde je to přidání do kolekce – metoda `add()`, vzhledem k možnému vzniku zablokování (viz. kapitola 4.7)

4.6.4 Atomicita a klíčové slovo volatile

Jako **atomickou akci** označujeme takovou akci, která probíhá prakticky najednou, nedělitelně. Takovou akci nelze přerušit během vykonávání, buď proběhne celá anebo vůbec. Zároveň do jejího ukončení nejsou vidět její výsledky. My již víme, že v Javě je spousta i jednoduchých úkonů rozdělených na více dílčích akcí a tudíž neatomických. Zápis a čtení z paměti je atomický pro všechny primitivní datové typy, s výjimkou `double` a `long`. Bohužel však i u primitivních datových typů může docházet k chybám v konzistenci paměti. V Javě 5 se spolu s novým paměťovým modelem objevuje i nová definice klíčového slova `volatile`.

Toto klíčové slovo zaručí nedělitelnost čtení i zápisu u všech primitivních datových typů a zároveň snižuje riziko chyb v konzistenci paměti. Ty nejsou vyloučeny, a to i přesto, že nedělitelné akce nemohou být prokládány kompilátorem a interpretem. Téměř úplné odstranění těchto chyb dosáhneme právě deklarováním proměnných jako `volatile`.

Vlastnosti volatile proměnných

- Jsou vždy zapsány do hlavní paměti, kompilátor je nesmí držet v registru
- Před čtením volatile proměnné vzniká read-barrier, po zápisu je provedena write-barrier
- Kompilátor při přerovnávání přístupů k paměti nesmí zasáhnout přes hranice přístupu k volatile proměnné
- Čtení a zápis volatile proměnné je vždy atomický
- Tvoří relaci „nastává-před“ – pokud při zápisu do volatile proměnné v jednom vlákně přijde z druhého vlákna požadavek na přečtení této hodnoty, proběhne čtení až po fyzickém zapsání proměnné a tato hodnota je tedy vždy z druhého vlákna správně viditelná

(Projekt 464_-_volatile-promenne)

```
public class AtomickyCitac
{
    volatile boolean priznak;

    void tryToDoSomething()
    {
        while (!priznak)
            doSomething();
    }

    void doSomething()
    {
        // nejaky kod
    }
}
```

Proměnnou `priznak` použijeme jako „flag“ – dokud něco neplatí, vykonávej určitý kód. Tím, že ji definujeme jako `volatile`, bude tento vzor bezpečný i při použití vláken.

4.6.5 Neměnné (immutable) objekty

Strategie neměnných (immutable) objektů se ukazuje jako velmi výhodná a užitečná při tvorbě paralelních aplikací. Neměnným objektem nazveme

takový objekt, jehož stav se již dále po vytvoření nemůže změnit. Díky tomu jej nemůže poškodit interference procesů, ani se jich netýká problém nekonzistence paměti. Samozřejmě, že režie při vytváření immutable objektů je o něco vyšší, než při práci s klasickými objekty, ale lze ji vcelku efektivně vyvážit.

Existuje několik pravidel, kterých je vhodné se držet při vytváření neměnných objektů:

- Žádné přístupové metody pro změnu datových složek („gettery“, „settery“)
- Všechny datové složky musí být deklarovány jako final a private
- Je zakázáno přepisování (override) metod, nejjednodušší je opět nastavit je jako final
- Zakázat změnu datových složek, pokud jsou jimi objekty, které to umožňují

4.7 Problémy s aktivitou

Jako aktivitu označíme schopnost paralelní aplikace fungovat bez zdržení. Samozřejmě, že vláknové programování nepřináší jen výhody a tvůrce aplikací se může setkat s několika problémy, vznikajícími při běhu aplikace. V této části se podíváme na tři nejdůležitější typy problémů s aktivitou, se kterými se můžeme setkat. Především je důležité se vyhnout situacím, při kterých mohou vznikat. Je jen na programátorovi, aby jim dokázal předejít.

4.7.1 Zablokování (deadlock)

Deadlockem rozumíme situaci, kdy jedno nebo více vláken je trvale zablokováno, protože na sebe vzájemně čekají. Jedno vlákno (nebo jeho metoda) nemůže být ukončeno, protože čeká na druhé vlákno. To je ale napsané stejně a čeká na první vlákno. Tím vzniká jakýsi začarovaný kruh a program (nebo jeho část) neběží. Univerzální řešení deadlocku samozřejmě

neexistuje, pokud k němu dojde, tak se jedná o chybu programátora a je také na něm, aby tuto chybu odstranil a přepsal aplikaci tak, aby vůbec nevznikla.

4.7.2 Odepření zdrojů (starvation)

Odepření zdrojů (starvation, doslova „vyhladovění“) je situace, kdy vlákno nedokáže získat přístup k prostředkům a díky tomu nemůže probíhat. Důvodem může být například synchronizovaná metoda, které dlouho trvá, než předá řízení zpět. Pokud jedno vlákno často tuto metodu volá, může ostatním zablokovat přístup k prostředkům. Vznik tohoto problému ukazuje na špatný návrh aplikace.

4.7.3 Vzájemné brzdění (livelock)

Obdoba deadlocku, nejedná se o zablokování vláken, pouze jsou vlákna zaneprázdněna reagováním na jiná vlákna a to jim znemožňuje činnost.

5 Knihovna java.util.concurrent

V předchozí kapitole jsme se zabývali pouze nízkoúrovňovým řešením, které nabízí API. Tato rozhraní byla součástí Javy od samého počátku, a postačují pro řešení jednoduchých úkolů. Současné nároky však sahají mnohem výše a při složitějších zadáních je nutné sáhnout k vyšší úrovni. Toto platí zejména pro rozsáhlé paralelní aplikace, které dokáží plně využít dnešní vícejádrové a víceprocesorové systémy.

V tomto oddíle se zaměřím na některé funkce a prostředky, které nabízí Java 5. Většina těchto funkcí je implementována v knihovně `java.util.concurrent`. Tyto struktury nepůjdou použít na starších verzích Javy, pro jejich správnou funkci je potřeba Java 5 a vyšší.

Knihovna `java.util.concurrent` je přizpůsobena pro co nejintuitivnější návrh paralelních aplikací tak, aby pro funkční a výkonnou aplikaci bylo třeba použít co nejméně kódu. A vzhledem k tomu, že všechny objekty a metody jsou navrženy velmi efektivně, je výkonnostní rozdíl oproti starším verzím Javy opravdu znát. Příkladem za všechny jsou např. atomické typy nebo kolekce přizpůsobené pro paralelní prostředí.

5.1 Atomické proměnné

Atomické třídy jsou novinkou Javy 5 a jsou přizpůsobeny pro práci v paralelních programech. Fungují jako jakési „obaly“ pro primitivní typy a zaštiťují jejich atomické zpracování v aplikacích. Všechny třídy obsahují metody `get()` a `set()` pro atomický zápis do volatile proměnných příslušného typu. Tedy metoda `set()` vytváří „nastává-před“ vztah mezi ní a všemi následujícími volání metody `get()`. Tím máme vždy zajištěnu aktuální hodnotu proměnné. Takto ošetřeny jsou i další metody z knihovny, my se blíže zaměříme na použití metody `compareAndSet()`. Knihovna dále obsahuje i jednoduché aritmetické metody pro práci s nedělitelnými proměnnými.

Balík `java.util.concurrent.atomic` obsahuje 9 tříd:

`AtomicInteger`

`AtomicIntegerArray`

`AtomicIntegerFieldUpdater`

Podle stejné konvence tři třídy pro `long`:

`AtomicLong`

`AtomicLongArray`

`AtomicLongFieldUpdater`

A stejně tak pro referenci:

`AtomicReference`

`AtomicReferenceArray`

`AtomicReferenceFieldUpdater`

Dále jsou v balíčku třídy

`AtomicBoolean`

`AtomicMarkableReference`

`AtomicStampedReference`

5.1.1 Práce s atomickými proměnnými

Třídy typu `AtomicXXX` tedy podporují atomické operace, což může být velmi výhodné. Ukažme si to na úloze typu „porovnej a nastav“ (Compare and Set). Budeme uvažovat aplikaci, ve které budeme potřebovat kód generující po sobě jdoucích ID. Nabízí se samozřejmě řešení pomocí synchronizovaných metod, kde budeme inkrementovat proměnnou typu `long` a vracet ji. Pro jednodušší aplikace s méně vlákny může takto implementovaný algoritmus fungovat, bohužel v náročném programu, kde bude spousta vláken rychle za sebou požadovat ID, stane se tato kritická sekce slabým místem aplikace a bude docházet ke zpomalení. Tato úloha je v praxi velmi často používaná a

s příchodem Javy 5 a atomických tříd pro práci s primitivními proměnnými je její řešení nejen velmi elegantní, ale i výkonné.

Na následujícím příkladu ukážeme implementaci metody, která bude vracet ID typu `long` vždy o jedničku vyšší, než bylo předchozí. Využijeme zde objekt typu `AtomicLong` a jeho metodu `compareAndSet(long a, long b)`. Druhou možností, jak toto řešit je použití funkčně velmi podobné metody `AtomicLong.getAndIncrement()` popřípadě metody `incrementAndGet()`. Řešení úlohy pomocí metody `compareAndSet()` je však podle mě názornější.

```
AtomicLong id;
public long getNextID()
{
   while (true)
   {
         long x = id.get();
         long pom = x++;
         if(id.compareAndSet(x, pom))
               return x;
   }
}
```

Do proměnné `x` uložíme poslední vygenerované ID a do proměnné `pom` další ID, které by mělo následovat. Pokud metoda `compareAndSet()` vrátí true, znamená to, že pracujeme s aktuální hodnotou ID a můžeme jej vrátit. Pokud je false, proběhne celý cyklus znovu, získáme aktuální hodnotu, a znovu porovnáváme. Celá akce je navíc díky použití objektu typu `AtomicLong` opravdu nedělitelná a máme zaručen správný výsledek, i slušnou efektivitu programu.

Na dalším příkladu si můžeme ukázat úpravu naší třídy `Counter` za použití objektu `AtomicInteger`. Díky využití této třídy se úplně můžeme vyhnout synchronizovaným metodám, které jsme používali v první modifikaci tohoto příkladu. Zároveň vidíme, že jsou zde implementovány metody pro inkrementaci a dekrementaci, o které víme, že rozhodně není atomickou

operací. Díky použití `AtomicInteger` tedy můžeme příklad jednoduše modifikovat a přitom předejít interferenci procesů i bez synchronizace.

```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicCounter
{
   private AtomicInteger i = new AtomicInteger(0);

   public void zvysit()
   {
         i.incrementAndGet();
   }

   public void snizit()
   {
         i.decrementAndGet()
   }

   public int vratHodnotu()
   {
         i.get();
   }
}
```

5.1.2 Atomická pole

`AtomicXXXArray` je objekt pro atomickou práci s polem. Předem můžeme říci, že plní stejnou funkci, jakou by zajistilo i pole `AtomicXXX` objektů, ovšem s výrazně nižší paměťovou režií. Je to pole, jehož položky (pozor, ale ne celé pole) můžeme atomicky modifikovat. Vyšší výkon je zde díky tomu, že pole atomických objektů bude obsahovat takový počet atomických objektů, kolik bude prvků pole. Oproti tomu `AtomicXXXArray` jsou v podstatě pouze dva objekty, samotné pole a obal.

5.1.3 Atomické změny členských proměnných třídy

Tuto funkci zajišťují objekty typu `AtomicXXXFieldUpdater`. Ty nám umožňují atomicky modifikovat členské proměnné třídy, které ovšem musí být deklarovány jako volatile a musí být modifikovány přes tento updater. V případě, že by tomu tak nebylo, bude režie na modifikaci proměnných o mnoho vyšší, než při použití těchto objektů.

Pozn.: Další novinkou v Javě 5 ohledně atomických změn členských proměnných jsou třídy `AtomicMarkableReference`, a `AtomicStampedReference`. Tyto dvě třídy umožňují atomickou změnu dvojice členských proměnných – reference a boolean v prvním případě, a reference a int v případě druhém. Zatím toto není příliš efektivně implementováno, avšak v budoucnu se počítá s rozšířením těchto tříd.

5.2 Rozhraní Lock

Klasické synchronized bloky kódu nabízejí jednoduchý typ zámku. Pro potřeby složitější a pokročilejší práce s objekty typu zámek nám knihovna `java.util.concurrent` nabízí rozhraní `Lock`. Objekt typu `Lock` funguje na podobném principu jako klasické zámky, v jeden okamžik může být `Lock` vlastněn pouze jedním vláknem. Nabízejí však více možností.

Největší výhoda oproti klasickým zámkům je v tom, že mohou vzít zpátky svůj pokus o získání zámku. Např. metoda `tryLock()` zruší pokus na základě timeoutu, metoda `lockInterruptibly()` v okamžiku, kdy jiné vlákno odešle přerušení ještě před získáním zámku. Díky těmto vlastnostem můžeme například bezpečně předcházet např. *livelocku* (kapitola 4.7.3).

Java také svazuje zámky s konkrétním blokem kódu, tudíž je nelze předávat např. do jiné metody. I toto odpadá s použitím objektu typu `Lock` a jeho metod `acquire()` a `release()`.

5.2.1 Výhody oproti synchronized bloku

Objekty typu Lock významně rozšiřují práci se zámky oproti blokům definovaným jako `synchronized`. Tato podkapitola se věnuje těm nejzásadnějším výhodám.

Zajímavou možností, kterou nabízí `Lock` je podpora tzv. hand over hand algoritmů, používaných například při práci se spojovými seznamy (datová struktura, kde prvek seznamu si pamatuje pouze odkaz na předchůdce a následovníka) – můžeme zamykat položky seznamu postupně (např. zamkneme nejdříve první a druhý prvek seznamu, odemkneme druhý, zamkneme třetí, apod.). Tuto akce nelze s klasickým `synchronized` blokem nelze provést.

Dále můžeme rozšířit podmínky pro čekání pomocí vytvoření objektů typu `Condition` (rozhraní ze třídy `java.util.concurrent.locks`), které nám umožní definovat více než jednu podmínku.

```
class BoundedBuffer {

final Lock lock = new ReentrantLock();
final Condition notFull  = lock.newCondition();
final Condition notEmpty = lock.newCondition();

final Object[] items = new Object[100];
int putptr, takeptr, count;

public void put(Object x) throws InterruptedException {
  lock.lock();
  try {
    while (count == items.length)
      notFull.await();
    items[putptr] = x;
    if (++putptr == items.length) putptr = 0;
    ++count;
    notEmpty.signal();
  } finally {
    lock.unlock();
  }
}
```

```
  public Object take() throws InterruptedException {
    lock.lock();
    try {
      while (count == 0)
        notEmpty.await();
      Object x = items[takeptr];
      if (++takeptr == items.length) takeptr = 0;
      --count;
      notFull.signal();
      return x;
    } finally {
      lock.unlock();
    }
  }
}
```

Tento příklad definuje zásobník a metody pro přidávání a ubírání. Pokud je zásobník prázdný, vlákno je zablokováno a čeká se na vložení prvku. Stejně tak dojde k zablokování vlákna s metodou `put()`, pokud je volána na plný zásobník. Díky objektu `Lock` můžeme provádět zablokování i čekání v obou vláknech. Důležité metody jsou `await()`, která čeká buď na vypršení timeoutu a nebo na přerušení, či signál. Ten mají na starosti metody `signal()` a `signalAll()`.

Z dalších možností nabízí zámky odvozené od Lock prostředky pro přerušení či nastavení timeoutu, po kterém se vlákno čekající bude pokoušet o probuzení. Můžeme vytvořit i tzv. **spravedlivé zámky**, kdy vláknům bude zámek přiřazován v pořadí, v jakém o něj požádala.

5.2.2 Reentrantní a nereentrantní zámky

Klasické `synchronized` bloky jsou z hlediska opakovaného vstupu do bloku **reentrantními** zámky. To znamená, že vlákno, které vlastní zámek, může do `synchronized` bloku stoupit znovu. Objekty typu `Lock` mohou být jak reentrantní tak **nereentrantní.** V tom druhém případu dojde k deadlocku a vlákno v podstatě zamkne samo sebe, pokud se opakovaně pokusí vstoupit do synchronizovaného bloku. Díky

`java.util.concurrent.locks.ReentrantLock` můžeme tyto typy zámků vytvářet programově.

5.3 Synchronizers (Synchronizační primitiva)

Knihovna `java.util.concurrent` nám nabízí i implementaci synchronizačních primitiv (kapitola 2.2.9), se kterými pak lze programově řídit běh programu.

5.3.1 Typ „semafor“

Semafor je typ synchronizačního primitiva, používaného pro správu volných prostředků a omezení počtu současně vykonávaných úkolů. Klasický semafor (někdy také nazývaný Dijkstrův semafor) má jeden celočíselný parametr, který reprezentuje počet volných prostředků. Dále jsou nad semaforem implementovány operace zvýšení a snížení počitadla prostředků.

Semafor v Javě je reprezentován třídou `Semaphore`. Konstruktor má celočíselný parametr, který určuje počet vláken, která mohou být v jeden okamžik v kritické sekci. Jakmile jedno vlákno uvolní slot, může jej získat hned jiné. Pokud je počitadlo semaforu 0 (žádné volné prostředky), řadí se čekající vlákna do fronty a jsou obsluhovány postupně hned po uvolnění prostředků.

Drobnou modifikací semaforu je třída `CountDownLatch`, která funguje na podobném principu, ale s tím rozdílem, že uvolněný slot již nikdo nezíská. Vlákna mohou pomocí `await()` počkat, než se vyčerpají všechny sloty a počitadlo klesne na nulu, poté je již zámek odemčen a každé další volání `await()` projde bez čekání.

```
class Semafor
{
   private Semaphore semaphore = new Semaphore(10, true);

   public Item getItem() throws InterruptedException
   {
     semaphore.acquire();
     return new Item();
   }

  public void putItem(Item x)
  {
     if (x.isUnused())
       semaphore.release();
  }
}
```

5.3.2 Typ bariéra

Pod tímto pojmem si můžeme představit analogii s klasickou bariérou jako prostředkem pro zastavení či zablokování. Bariéra v Javě funguje tak, že pokud nějaké vlákno požádá o bariéru, je tak zablokované do doby, než se na bariéře sejdou všechna ostatní vlákna. Pokud se tak stane, máme k dispozici objekt typu `Runnable`, který nám umožní vykonat nějakou akci a až poté jsou vlákna odblokována a mohou pokračovat.

Vytvoříme objekt `CyclicBarrier`, jako parametry předáme konstruktoru počet vláken, kterých se bude bariéra týkat a volitelně objekt typu `Runnable`, který se vykoná po doběhnutí vláken k bariéře. Pak už jen v kódu vlákna zavoláme metodu `await()` nad objektem typu `CyclicBarrier`.

(Projekt 523_-_cyklicka-bariera)

5.3.3 Exchanger

Exchanger umožňuje dvěma vláknům vyměnit si v určitém okamžiku data a zapsat je do paměti, přičemž systém se postará o správné uložení dat. Největší význam má použití Exchangeru pro zabránění chybám v konzistenci paměti – při použití tohoto objektu je zajištěn i vznik relace „nastává-před“. Samotná výměna dat probíhá pomocí metody `exchange()`.

```
Exchanger<DataType> exchanger = new Exchanger<DataType>();
DataType item1;
DataType item2;
echnager.exchange(item1);
```

Metoda počká, dokud nedorazí druhé vlákno a vymění si s ním objekty.

5.4 Exekutory

V jednoduchých aplikacích existuje silná vazba mezi úkolem, který provádí `Runnable` a tím, jak je definováno vlákno `Thread`. Tento princip je vyhovující pro menší programy, avšak v případě, že vytváříme velký projekt, je vhodnější oddělit správu a vytváření vláken od zbytku aplikace. Objekty, které nám toto umožní pak v Javě vytváříme jako objekty implementující rozhraní `Executor`. Exekutor je služba, která nám umožní vykonat úkol, který jí předáme jako `Runnable`. Toto je možné udělat i typicky v jiném vlákně, popřípadě skupině vláken (`ThreadPool`) – viz. kapitola 5.4.4.

V balíčku `java.util.concurrent` najdeme tři rozhraní, chovající se jako exekutor.

- **`Executor`**
  Umožňuje spouštět nové úkoly a vlákna.

- **ExecutorService**

  Jedná se o podrozhraní rozhraní `Executor` a umožňuje nám řídit životní cyklus jednotlivých úkolů i vlastní spouštění nových úloh.

- **ScheduledExecutorService**

  Opět rozšíření předchozího `ExecutorService`, umožňuje spouštět úlohy pravidelně.

5.4.1 Rozhraní Executor

Toto rozhraní poskytuje jedinou metodu `execute()`, která funguje jako alternativní možnost vytvoření vlákna. Pokud vytváříme vlákno klasicky, tedy tak, že konstruktoru objektu `Thread` předáme objekt typu `Runnable` a zavoláme metodu `start()`, můžeme pomocí exekutoru tuto konstrukci nahradit tímto způsobem:

Klasický způsob:

```
(new Thread(r)).start();
```

Pomocí exekutoru:

```
e.execute(r);
```

Kde `r` je objekt typu `Runnable` a `e` objekt typu `Executor`.

Definice metody `execute()` nám dává více možností v práci s vlákny, a jejich ovládání. Klasické vytvoření vlákna probíhá tak, že nejprve vlákno definujeme a to je prakticky ihned spouštěno. V závislosti na implementaci může metoda `execute()` udělat to samé, anebo spustit vlákno buď se zpožděním nebo jej umístit například do skupiny vláken (`ThreadPool`).

5.4.2 Rozhraní ExecutorService

Vychází z rozhraní `Executor`, ale doplňuje navíc metodu `submit()`, která je podobná jako `execute()`, nabízí však všestrannější možnosti. Může totiž přijímat nejen objekty typu `Runnable`, ale zároveň i objekty typu `Callable`.

Rozhraní Callable

`Callable` je rozhraní, které je v podstatě shodné s `Runnable`, ale navíc oproti němu umí vrátit hodnotu. Stejně jako `Runnable` je navrženo pro třídy, jejichž instance budou spouštěny jinými vlákny. Obsahuje jedinou metodu `call()`, která vrátí výsledek výpočtu, popřípadě vyhodí výjimku. V tom je největší výhoda `ExecutorService` oproti klasickému `Execute`.

Rozhraní Future

Metoda `submit()` vrací objekt typu `Future`, který používáme pro načtení návratové hodnoty objektu typu `Callable` a v podstatě jakéhokoliv asynchronního výpočtu. Poskytuje metody, které nám umožní zjistit, zda výpočet proběhl v pořádku, případně počkat na jeho dokončení a zkompletovat výsledek. Výsledek vrací pomocí metody `get()`, zároveň umožňuje stornování procesu metodou `cancel()`. Dále nabízí booleovské metody pro zjištění toho, zda byl výpočet zrušen či dokončen.

Výhodou `ExecutorService` je i to, že nabízí metody pro své vlastní ukončení (popř. ukončení nadřazeného objektu typu `Executor`). Volání `shutdown()` způsobí, že budou ukončeny současně běžící úkoly a nové už nebudou spuštěny. `shutdownNow()` způsobí okamžité ukončení všech běžících úloh. Samozřejmě ale musí být správně implementována podpora pro přerušení (např. správné vyhození výjimky `InterruptedException` – viz kapitola 4.4)

5.4.3 Rozhraní ScheduledExecutorService

Přidává k metodám svých nadřazených rozhraní (`ExecutorService` a jemu nadřazené `Executor`) metodu `schedule()`, umožňující spuštění úkolu `Runnable` nebo `Callable` po uplynutí určité doby. Zároveň metoda vrací objekt typu `ScheduledFuture`, se kterým pak pracujeme podobně, jako s klasickým `Future`.

Oproti metodě `execute()`, která je vždy spouštěna okamžitě, je výhodou `ScheduledExecutorService` právě oddálení spuštění této metody. Doba oddálení je předávána jako parametr metodě `schedule()` zároveň se specifikací jednotky času. Tyto hodnoty jsou ovšem relativní vzhledem k době spuštění. Jednoduchým trikem však můžeme naplánovat spuštění přesně v daný čas. Vytvoříme objekt typu `java.util.Date`, který inicializujeme jako požadovaný čas spuštění a uděláme rozdíl časů, čímž získáme požadovanou dobu spuštění.

```
    scheduler.schedule(task, date.getTime() -
System.currentTimeMillis(), TimeUnit.MILLISECONDS)
```

Rozhraní ještě poskytuje dvě metody pro periodické spouštění úkolů. Jsou to metody `scheduleAtFixedRate()` a `scheduleAtFixedDelay()`. Rozdíl je v tom, že `scheduleAtFixedDelay()` čeká na dokončení předchozího běhu úlohy a teprve pak spouští úlohu znovu (po uplynutí periody), zatímco `scheduleAtFixedRate()` spouští úlohu periodicky a vždy s pevným časovým rozestupem.

(Projekt 54_-_prace-s-executorem)

5.4.4 Skupiny vláken (Thread Pool)

V předchozím textu jsme se několikrát dotkli termínu „skupina vláken“ (Thread Pool). Většina tříd implementujících rozhraní `Executor` a jemu podřízené pracuje právě se skupinami vláken, které zahrnují tzv. pracovní vlákna (worker threads). Tento typ existuje odděleně od rozhraní `Runnable` a `Callable` a slouží ke spouštění více úkolů. Skupiny vláken minimalizují zároveň režii spojenou s vytvářením vláken – vytvoření a správa vláken je náročnější operace a u velkých projektů může znamenat ztrátu výkonu programu. Proto se při navrhování větších projektů pracuje spíše se skupinami vláken vzhledem k náročnosti aplikace, která souvisí s vytvářením mnoha objektů a přidělováním paměti těmto objektům.

Nejpoužívanějším objektem pro správu skupiny vláken je tzv. `FixedThredPool`, který si můžeme představit jako skupinu s pevným počtem vláken. Tato skupina vždy obsahuje konkrétní počet vláken, pokud existuje větší počet vláken, ostatní vlákna čekají ve frontě na uvolnění místa. Pokud je vlákno ve skupině ukončeno, okamžitě jej nahradí jedno čekající. Výhoda tohoto „fondu“ vláken spočívá v tom, že program je schopen pružněji reagovat na vytížení a zároveň za cenu nižších systémových prostředků. Zároveň takto můžeme kontrolovat vytížení systému a regulovat tak jeho práci. Zjednodušeně řečeno – v případě hodně náročné aplikace, která bude vytvářet spousty vláken, může dojít ke stoprocentnímu vytížení systému a tím také zamrzání našeho programu a znemožnění práce se systémem. Pomocí skupin vláken tomu můžeme zamezit a ošetřit program, aby si bral jen určitou část systémových prostředků.

Jako teoretický příklad nám může sloužit aplikace webového serveru. Kvůli paralelismu musí každý http požadavek zpracovávat v samostatném vlákně. Pro každý požadavek vytváří samostatné vlákno, což je v pořádku až do doby, kdy režie přesáhne volné prostředky. Pak systém přestane reagovat a program se významně zpomalí. Řešení je jednoduché – vytvoříme skupinu

vláken s pevným počtem vláken, která budou v daném okamžiku aktivní a obsloužena. Sice zaplatíme tím, že aplikace nebude obsluhovat všechny požadavky v reálném čase a bude je uchovávat ve frontě, než na ně přijde řada. Nedojde však k zahlcení systému, které v tomto případě může mít horší dopad, než jen pozdržení zpracování požadavku.

Objekt typu `Executor`, který umí pracovat se skupinami vláken, vytvoříme zavoláním metody `newFixedThreadPool(int)` ze třídy `Executors` (pozor na „s" na konci, nejedná se o rozhraní `Executor`, o kterém byla řeč dříve). Tato třída poskytuje tzv. **výrobní metody** pro vytváření skupin vláken.

5.4.4.1 Rozhraní ThreadFactory

Pro práci s třídou `Executors` ještě nejprve považuji za nutné zmínit rozhraní `ThreadFactory`, se kterým výrobní metody třídy `Executors` pracují. Toto rozhraní slouží pouze k jedné věci, a tou je vytvoření vlákna „na požádání". Obsahuje jedinou metodu `newThread(Runnable r)`, která vrací objekty typu `Thread`. Důvod pro použití je jednoduchý – náš program může používat vlákna mnohem flexibilněji, pracovat s podtřídami vláken apod. Nejjednodušší implementací této metody je samozřejmě jednoduché zavolání `return new Thread(r);` kde `r` je objekt typu `Runnable`. Více možností ovšem dostáváme, pokud použijeme třídu `Executors` a její metody, které mají pokročilejší implementaci práce s `ThreadFactory`. Jako příklad můžeme uvést např. statickou metodu `defaultThreadFactory()` vracející objekt typu `ThreadFactory`, umožňující vytvářet vlákna pomocí objektu typu `Executor` v rámci jedné skupiny vláken. Rozšířením metody `defaultThreadFactory()` je metoda `privilegedThreadFactory()`, která navíc umožňuje bezpečnostní kontrolu kontextu.

5.4.4.2 Metody třídy Executors

Výrobní metody třídy `Executors` nám tedy umožňují pokročilou práci se skupinami vláken. Dále nám umožňují vytvářet tyto skupiny co nejefektivněji díky mnoha metodám, které tato třída poskytuje. Většina těchto metod nám vrací objekty typu `ExecutorService`, `ScheduledExecutorService` nebo `Callable` (viz kapitola 5.4).

Z těch nejdůležitějších metod zmíním výrobní metody

```
newCachedThreadPool()
newSingleThreadExecutor()
newFixedThreadPool()
```

- **`newCachedThreadPool()`**

  Vytvoří skupinu vláken, ale zároveň dokáže v rámci skupiny pracovat již s vytvořenými vlákny. Tento objekt je vhodný pro aplikace, které spouštějí mnoho krátkodobých úkolů.

- **`newSingleThreadExecutor()`**

  Vytváří objekt typu `ExecutorService`, který však spouští úlohy po jedné z fronty čekajících.

- **`newFixedThreadPool()`**

  Vznikne objekt typu `ExecutorService`, který dokáže v jednu chvíli obsloužit pevný počet vláken.

Obdobné metody existují i pro `ScheduledExecutorService`.

Pokud nestačí tyto třídy, můžeme skupiny vláken vytvářet i pomocí tříd `ThreadPoolExecutor` a nebo `ScheduledThreadPoolExecutor` z balíčku `java.util.concurrent`. Ty nabízí opravdu pokročilé prostředky pro práci se skupinami vláken.

Na následujícím příkladu si ukážeme jednoduchý projekt pro práci nejen se skupinami vláken, ale zároveň i objekty typu `Executor`. Budeme tedy předpokládat, že máme program, který vyžaduje vykonávání mnoha malých operací současně s tím, že každá operace bude běžet ve svém vlastním vlákně. V této části jsme si řekli, že pokud by bylo vláken mnoho, může dojít k nadměrnému zatížení systému. Tomu předejdeme vytvořením fixní skupiny vláken s pevným počtem vláken obsluhovanými v jeden okamžik. V tomto čistě teoretickém příkladě nejprve vytvoříme skupinu vláken, o spuštění se nemusíme starat, protože to má na starosti objekt typu `ExecutorService` a nakonec je „inteligentně" vypneme – pomocí metod `shutdown()` a `awaitTermination()` objektu typu `Executor` počkáme na dokončení.

```
ExecutorService executor = Executors.newFixedThreadPool(10);
for (int i = 0; i < 100; i++)
      {
        executor.execute(new Runnable()
        {
          public void run()
          {
             // zde vykoname nejaky kod
          }
        }
        );
      }
      executor.shutdown();
      executor.awaitTermination(60, TimeUnit.SECONDS);
```

5.5 Paralelní kolekce

Kolekce a práce s nimi jsou silnou stránkou Javy. Klasické kolekce ovšem naráží na nepřipravenost pro práci ve vícevláknové aplikaci. To, že je Java tak mocným jazykem, je hlavně díky rozsáhlým knihovnám API. A protože knihovna `java.util.concurrent` vznikla hlavně z důvodů co nejvíce usnadnit práci s vícevláknovými programy a implementovat nejzásadnější věci tak, aby programátor použil už hotové řešení, najdeme i zde třídy a rozhraní připravené pro nasazení ve vícevláknových aplikacích. Doplňují hlavně atomicitu operací a pomáhají předcházet chybám v konzistenci paměti,

protože definují relaci „nastává-před“ mezi operací přidání do kolekce, a operací čtení nebo odstranění z kolekce.

Tyto doplňky definují práci s `Queue`, `List`, `Map` a `Set`. Ty jsou navíc dále rozloženy do dalších tříd a rozhraní.

5.5.1 Fronta (Queue)

Fronta (FIFO) je datová struktura, založená na principu „First in, first out“. To znamená, že prvek, který do ní byl jako první vložen, bude také jako první odebírán, nebo také jinak řečeno – prvky jsou vkládány na konec fronty, a odebírány ze začátku. Implementaci této struktury zajišťuje v Javě rozhraní `java.util.Queue`. My se v této části zaměříme na implementaci tohoto rozhraní v třídách knihovny `java.util.concurrent`. Najdeme zde dvě rozhraní, která definují frontu a třídy, které tato rozhraní implementují. Těmito dvěma rozhraními jsou `BlockingQueue` a `BlockingDequeue`. Obě tato rozhraní mají společnou vlastnost – blokují nebo jiným způsobem omezují časovým limitem pokusy o přidání prvku do plné fronty nebo odebírání prvku z prázdné fronty a stejně tak nám umožňují vytvořit frontu s omezenou kapacitou.

5.5.1.1 Rozhraní BlockingQueue

Rozhraní `BlockingQueue` rozšiřuje klasickou frontu `Queue`, a připravuje ji pro použití v prostředí vícevláknových aplikací. Blokující fronta neumožňuje vložit *null*ový element a může být kapacitně omezena. Pokud neurčíme její kapacitu, nastaví se automaticky na konstantu `Integer.MAX_VALUE`. Důležitou vlastností je čekání – jak uvádím výše. Operace počká, pokud chceme vkládat do plné fronty (nebo číst z prázdné) až do doby, kdy tato činnost bude moci proběhnout bezpečně.

Všechny implementace rozhraní `BlockingQueue` jsou vláknově bezpečné (thread-safe), téměř všechny metody podporují atomické operace a disponují nějakým druhem vnitřního zámku. Třídy implementující toto rozhraní se nejčastěji používají pro simulování úlohy **producent-konzument**. Chybí však jedna věc – signalizace „closed“ (fronta je uzavřena a žádný prvek už producent nebude přidávat). Je tak na programátorovi, aby tuto implementaci vytvořil sám (např. přidáním prvku ve speciálním tvaru na konec fronty a jeho kontrolou). Zajímavou vlastností je i to, že `BlockingQueue` zároveň implementuje rozhraní `Collection`, takže lze použít metody i z tohoto rozhraní, obecně se to však kvůli efektivitě nedoporučuje.

Rozhraní tedy v sobě spojuje práci s frontou a kolekcí, čímž máme větší možnosti, ale zároveň je třeba vhodně volit, jaké metody použít. Například pro vkládání do fronty máme k dispozici metody `boolean add()`, `boolean offer()` a `void put()`. Na první pohled dělají to samé, avšak není to až tak úplně pravda. Funkčně blízké si jsou metody `add()` a `offer()`, obě zkusí okamžitě vložit prvek do fronty, ale `add()` vyhodí výjimku, pokud neuspěje, zatímco `offer()` vrátí méně agresivní `false`. Metoda `put()` počká, dokud se v plné frontě neuvolní místo a až pak prvek vloží. Stejně tak si jsou funkčně podobné metody `poll()` a `take()`. Metoda `take()` v případě neúspěchu čeká, zatímco `poll()` má specifikovaný timeout a po jeho uplynutí vrací *null*.

Nyní se zběžně podíváme na třídy implementující toto rozhraní. Obecně je můžeme rozdělit na třídy ve tvaru `XXXBlockingQueue` a ty ostatní.

5.5.1.1.1 Třídy ve tvaru XXXBlockingQueue

Tyto třídy nám umožňují čekat na položku (s timeoutem nebo bez).

- **`ArrayBlockingQueue`**
  Klasická seřazená fronta, jako první element ve frontě je ten, který je v ní

nejdelší dobu, jako poslední naopak prvek, který byl naposledy přidán. Tato fronta má však pevnou velikost (díky tomu je efektivní), při pokusu o přidání prvku do plné fronty dochází k blokování. Zároveň umožňuje nastavit “spravedlivý přístup” vláken ke frontě.

- **`LinkedBlockingQueue`**

  Implementace fronty jako spojového seznamu, můžeme ji nastavit kapacitu, ale tím přicházíme o největší výhodu, kterou je právě dynamická velikost. Většina operací nad touto frontou probíhá v lineárním čase.

- **`PriorityBlockingQueue`**

  Rovná prvky podle přirozeného třídění (tak, jak jsou ve třídách implementovány metody `compareTo()`)

5.5.1.1.2 Ostatní třídy implementující rozhraní BlockingQueue

- **`DelayQueue`**

  Reprezentuje frontu, ze které je položky možné odebírat až po uplynutí určité prodlevy (timeoutu), její elementy musí implementovat rozhraní `java.util.concurrent.Delayed`. Prvky jsou řazeny podle vypršení prodlevy – první ve frontě je prvek, kterému prodleva uplynula před nejdelším časem. Prvky, kterým prodleva ještě plyne, nemohou být odebrány, přestože se započítávají do počtu prvků ve frontě.

- **`SynchronousQueue`**

  `SynchronousQueue` je zvláštní případ fronty, která má nulovou kapacitu. Nesmí se do ní vkládat prvek, pokud už se jiné vlákno nepokouší o čtení z této fronty a naopak. Vlákna na sebe případně vzájemně čekají.

5.5.1.2 Rozhraní BlockingDequeue

`Dequeue` (`java.util.Dequeue`) je rozhraní, které také rozšiřuje klasickou frontu, stejně jako `BlockingQueue`. Je podrozhraním rozhraní `Queue` a přidává jednu důležitou vlastnost, kterou je **oboustrannost fronty**. Dequeue vlastně znamená „double-ended queue“, tedy fronta se dvěma konci. Oproti klasické frontě tak má metody pro práci se začátkem i koncem fronty. Toto rozhraní je podrozhraním `BlockingQueue`, takže od něj přebírá všechny klíčové vlastnosti – tzn. vláknová bezpečnost, nedovoluje *null*ové elementy a může mít omezenou kapacitu.

Metody, které poskytuje toto rozhraní, jsou rozšířením metod z rozhraní `BlockingQueue`. Metody pro práci s frontou jsou definovány tak, že pro všechny existuje nějaké ošetření toho, zda se operace povedla. Buď je metoda definována jako `boolean` a nebo `void`, ale se specifikovaným timeoutem, popř. vyhozenou výjimkou.

Vhodným testováním návratových hodnot metod definovaných jako `boolean` a zároveň odchytáváním výjimek tak můžeme předejít všem chybám při práci s oboustrannou frontou.

Jedinou implementací rozhraní `BlockingDequeue` je třída `LinkedBlockingDeque`. U této třídy bych zmínil zajímavou metodu `int drainTo()`, která umí vzít všechny prvky ve frontě a převést je do kolekce. Návratová hodnota je počet přenesených prvků.

(Projekt 551_-_producent-konzument)

5.5.2 Seznam (List)

Za seznam považujeme datovou strukturu bez pevně dané velikosti a obsahující prvky daného typu. Seznam je uspořádaná kolekce, která se do velké míry chová stejně jako pole, ale nemá fixní velikost. K jednotlivým prvkům můžeme přistupovat stejně jako v poli pomocí celočíselných indexů a efektivně seznam prohledávat. Seznam umožňuje vkládání duplicitních elementů a kromě indexového přístupu k prvkům umožňuje i klasickou iteraci díky rozhraní `Iterable`. Stejně jako u předchozí fronty existuje i pro seznam v knihovně `java.util.concurrent` jeho implementace pro lepší použití ve vícevláknovém prostředí. Na rozdíl od fronty zde najdeme pouze jednu implementaci, a to ve třídě `CopyOnWriteArrayList`.

5.5.2.1 Třída CopyOnWriteArrayList

Třída je implementací seznamu v knihovně `java.util.concurrent`. Je vláknově bezpečná a opět umí předcházet chybám v konzistenci paměti. Toho dosáhneme pomocí speciálně připraveného iterátoru a zároveň je tato implementace velmi efektivní. Důležité je, že iterátor po vytvoření zachází se seznamem jako s polem a neumožní v něm žádné změny, dokud neskončí iterátor. Tím máme zaručenu konzistenci dat a iterátor neumožní přidávání, odebírání ani změnu prvků v kolekci (poli). Proto nepotřebujeme žádnou synchronizaci. Všechny metody v podstatě odpovídají klasickému seznamu, musíme si pouze dávat pozor na práci s iterátorem.

5.5.3 Slovník/Mapa (Map)

Slovník (`Map`) je datová struktura, uchovávající data jako uspořádané dvojice ve tvaru klíč, hodnota. V podstatě můžeme říci, že mapujeme hodnotu na klíč. Výraz „slovník" používám kvůli zpětné kompatibilitě, rozhraní `java.util.Map` nahradilo rozhraní `Dictionary`, používané ve starších verzích API. Na rozhraní map můžeme pohlížet ze tří různých pohledů – jako

na množinu klíčů (neumožňuje duplicitní hodnoty klíčů), kolekci hodnot (hodnoty se mohou opakovat) a nebo na množinu uspořádaných dvojic klíč, hodnota (tato dvojice musí být unikátní – jedna stejná hodnota může být přiřazena více klíčům, jeden klíč však nesmí mít dvě různé hodnoty). Tato datová struktura je reprezentována rozhraním `java.util.Map`, některé její implementace uchovávají prvky např. seřazené (`TreeMap`). Nás budou ale opět zajímat pouze implementace tohoto rozhraní v knihovně `java.util.concurrent`.

Zde najdeme dvě rozhraní odvozená od `java.util.Map` a to `ConcurrentMap` a `ConcurrentNavigableMap`. `ConcurrentMap` je rozhraní definující atomické operace a zároveň vytvářející relaci „nastává-před“. `ConcurrentNavigableMap` tyto možnosti rozšiřuje ještě o prostředky pro vyhledávání – `NavigableMap` je rozhraní, definující metody pro vrácení nejbližších hodnot pro hledaný cíl.

5.5.3.1 Třída ConcurrentHashMap

Třída implementující rozhraní `ConcurrentMap` je přizpůsobena pro použití ve vícevláknových aplikacích a můžeme na ní pohlížet jako na hashovací tabulku (`java.util.HashTable`). Obsahuje tak metody, které korespondují metodám z `HashTable`. Všechny operace jsou sice vláknově bezpečné, ale neexistují zde zámky, nemáme tudíž možnost zamknout prvky mapy nebo celou mapu. Máme tedy vláknově bezpečnou implementaci, ale ne plně synchronizovanou tak, jak je tomu u jiných tříd.

Operace zápisu a čtení, prováděné nad tabulkou, tak mohou reflektovat jiná data, než jsou v aktuálním okamžiku k dispozici. Čtení získává data z posledního stavu tabulky (po dokončení posledního zápisu). Stejně tak iterátor reflektuje stav mapy v okamžiku vytvoření iterátoru a nebere v potaz změny, které nad tabulkou probíhají během existence iterátoru.

Můžeme také definovat tzv. **concurrency level**, což můžeme chápat jako stupeň nebo úroveň toho, do jaké míry bude mapa pružná v paralelním prostředí. HashMap je interně segmentována, čímž ovlivňuje počet souběžných změn mapy, zároveň je tato segmentace v podstatě náhodná a vzhledem k náhodnému umísťování dat v mapě se stupeň segmentace může lišit pro každou mapu. Tím, že stupeň segmentace určíme, říkáme tím vlastně, kolik vláken může současně měnit data v mapě. Pokud tento level určíme vhodným způsobem, můžeme získat nárůst výkonu, ale zároveň při špatném odhadu concurrency levelu budeme zbytečně plýtvat prostředky.

Metody odpovídají těm, které jsou použity v rozhraní Map, navíc je zde metoda putIfAbsent(), která vkládá prvek pouze za předpokladu, že v mapě ještě není. Je to atomický ekvivalent k následujícímu kódu

```
if (!map.containsKey(key))
       return map.put(key, value);
   else
       return map.get(key);
```

5.5.4 Množina (Set)

Množina je neindexovatelná kolekce, která zároveň obsahuje kontrolu přítomnosti prvku – každý prvek může být v množině pouze jednou. Musíme tak při implementaci množiny definovat třeba i ošetření události, kdy prvek množiny změní hodnotu na takovou, kterou už množina obsahuje. Z hlediska API je množina implementací rozhraní java.util.Set. V podstatě tedy řekneme, že množina nesmí obsahovat dva prvky e1 a e2, pokud platí e1.equals(e2).

V knihovně java.util.concurrent najdeme také paralelní implementaci rozhraní Set, a to ve třídě CopyOnWriteArraySet. Implementace této třídy spočívá v tom, že množina interně používá pro všechny své operace objekt typu CopyOnWriteArrayList (kapitola

5.5.2.1), sdílí tedy všechny základní vlastnosti včetně vláknové bezpečnosti. Je navržena pro operace, kde používáme málo prvků, které chceme často číst, ale málokdy měnit. Operace přidání nebo odebrání jsou totiž dost náročné vzhledem ke kopírování celé kolekce do pole. Oproti tomu iterátor pro procházení množinou je velmi rychlý.

6 Testování a výkon vícevláknových aplikací

V předchozí části jsme si na teoretické úrovni popsali prostředky, které Java nabízí pro vývoj vícevláknových aplikací. Všechny třídy i rozhraní API jsou navrženy s maximální možnou pozorností a tak, aby nabízela co nejširší pokrytí pro potřeby vznikajících programů. Samozřejmě ale stále platí, že sebevýkonnější funkce mohou být kontraproduktivní, pokud je aplikace špatně navržena a naprogramována. Proto velmi důležitou součástí vývoje vícevláknových aplikací je i testování výkonu a aktivity (kapitola 4.7). V této části práce se zaměříme na některé případy selhání aktivity a na to, co můžeme udělat pro to, aby k němu nedocházelo.

6.1 Problémy s aktivitou

O problémech s aktivitou jsme se zmínili již v kapitole 4.7 a nyní se tuto část pokusím rozvést. Klasickým (a nejčastějším) příkladem problému s aktivitou aplikace je **deadlock**, kterému bude věnována podstatná část této kapitoly.

6.1.1 Deadlock

Jak jsem uvedl dříve, jedná se o situaci, kdy se vlákna dostanou do nekonečné smyčky a jedno čeká na druhé, zatímco druhé vlákno v tu samou chvíli čeká na dokončení prvního. Pokud tato situace nastane, dochází k nekonečnému zacyklení a aplikace se stává nepoužitelnou. Nejznámějším učebnicovým příkladem pro deadlock je tzv. „problém hladových filosofů“, kde kolem stolu sedí pět filosofů a na stole je mísa rýže a pět hůlek, umístěných vždy mezi filosofy. Filosof může buď přemýšlet nebo jíst. K jídlu potřebuje dvě hůlky, vezme si hůlku po levé straně a čeká, jestli je volná hůlka i po pravé. K problému dojde v okamžiku, kdy po hůlce nalevo sáhne najednou všech pět filosofů. Neřešitelná situace, která nemůže nikdy skončit. Stejně je tomu i při špatné interferenci vláken.

Na rozdíl od databázových aplikací Java nemá na nižší úrovni prostředky pro detekci deadlocku a případné ukončení zacykleného programu, je tedy na vývojáři (a hlavně návrháři), aby k této situaci vůbec nemohlo dojít. To jde ovšem v ruku v ruce s kvalitním testováním aplikací, protože k deadlocku nemusí nutně dojít při každém spuštění „postižené“ aplikace.

6.1.1.1 Problém - špatné použití synchronizace a zámku (křížení)

Velmi častou příčinou deadlocku je špatná práce se synchronizací a zámky. Na následujícím příkladu jednoduché šablony ukážeme, jak dostat aplikaci do stavu deadlocku.

```
public class JednoduchyDeadlock {
    private final Object jedna = new Object();
    private final Object dva = new Object();

    public void prvniMetoda()
    {
        synchronized (jedna)
        {
            synchronized (dva)
            {
                udelejNeco();
            }
        }
    }

    public void druhaMetoda()
    {
        synchronized (dva)
        {
            synchronized (jedna)
            {
                udelejNecoJineho();
            }
        }
    }

    void udelejNeco()
    {
    }

    void udelejNecoJineho()
    {
    }
}
```

Pokud jedno vlákno zavolá metodu `prvniMetoda()` a druhé vlákno zároveň metodu `druhaMetoda()`, dojde k deadlocku, protože klíčové slovo `synchronized` způsobí vznik vnitřních zámků (kapitola 4.6.1) a běh aplikace se dostane do nekonečného cyklu. Zde je chyba ve špatném návrhu a v křížení pokusu o získání zámku. Pokud by synchronizace probíhala postupně, k deadlocku nedojde. Tím, že ale z jedné metody zároveň zamykáme i druhou, nastane tato situace. Hlavní zásadou tak je psát kód průhledně, dělit jej do logických celků a nekřížit je mezi sebou, což je ostatně i jeden z nosných prvků celého objektového programování. V rozsáhlých aplikacích může docházet k tisícům operací typu „zamkni-uvolni", stačí aby jediná z těchto operací selhala a aplikace je okamžitě uzamčena a nefunkční.

Podobná situace nastává v okamžiku, kdy vlákna spolupracují a není zde křížení, ale dojde k chybnému zamčení. Zejména, pokud existuje nějaké omezení počtu vláken nebo pokud špatně implementujeme semafor, či jiné synchronizační primitivum.

6.1.1.2 Problém - čekání na zdroje

Ačkoliv je křížení nejčastější příčinou deadlocku, není příčinou jedinou. Chtěl bych zmínit ještě jednu situaci, při které může dojít k deadlocku a tou je čekání na zdroje. Uvažujme například dvě databáze a vlákna, která k nim budou přistupovat. Může nastat situace, kdy vlákno bude chtít přistupovat k oběma databázím, připojí se k první databázi a bude čekat na připojení k druhé databázi, zatímco druhé vlákno bude naopak připojeno k druhé databázi a čekat na první. Nejedná se tedy o přímou interakci vláken, ale zdrojů. Řešením této situace je cyklické střídání vláken nebo skupin vláken tak, aby na každý potřebný počet spojení připadalo maximálně jedno vlákno. Tento problém je podobný odepření zdrojů (kapitola 4.7.2), **nejedná** se však o identickou situaci.

6.1.1.3 Předcházení a detekce deadlocku

Pokud program požaduje maximálně jeden zámek najednou, je situace jednoduchá a deadlock nikdy nemůže vzniknout. Problém nastává ve chvíli, kdy je těch zámků více. Jak jsem již zmínil, neexistuje univerzální řešení deadlocku, jediná obrana je ve správném návrhu aplikace. Proto na ochranu proti deadlocku musí být pamatováno už **při návrhu programu**. Pokud už musí program používat více zámků, mělo by to být opravdu jen nutné minimum. Každý zámek navíc zvyšuje potenciální riziko deadlocku. Zároveň je nutná komplexní analýza jak návrhu, tak zdrojového kódu. Je důležité správně analyzovat a odhadnout potenciální místa vzniku deadlocku a zároveň provést pečlivé testy s využitím debuggeru nebo profileru, abychom se ujistili, že naše data jsou konzistentní a nemůže dojít k žádné interferenci.

Dalším způsobem, jak minimalizovat riziko deadlocku, je využít toho, co nám nabízí balíček `java.util.concurrent.locks` a v našem projektu používat instance tříd implementujících rozhraní `Lock` a metodu `tryLock()`, která nám umožňuje specifikovat timeout zámku. Tím zabráníme nekonečnému zacyklení aplikace. Pokud metoda `tryLock()` vrátí `false`, víme, že došlo k nějaké chybě, nemusí nás zajímat jaké a můžeme implementovat ošetření této chyby. Například tím, že uvolníme zámek, opakujeme pokus za několik okamžiků, v případě opakovaného selhání ukončíme vlákna, která by mohla problém způsobovat a uvedeme program do konzistentního stavu. Tato technika je však použitelná pouze pro málo vláken (ideálně dvě) a jedná se o nouzové řešení. Aplikace by totiž měla být připravena a navržena tak, abychom tuto techniku vůbec nemuseli používat.

V případě, že používáme klasické `synchronized` zámky, může nám pomoci i tzv. **výpis vláken** (thread dump), který umí vrátit Java Virtual Machine. Tento výpis funguje i s objekty typu `Lock`, ale neposkytne tolik informací. Na Windows jej získáme stisknutím *Ctrl* + *Break*.

6.1.2 Ostatní selhání aktivity

Už jen krátce zde zmíním některé situace, kdy může dojít k jinému selhání aktivity, než je deadlock.

6.1.2.1 Odepření zdrojů (Starvation)

K „vyhladovění“ dojde, pokud je vláknu trvale odepírán přístup k datům, která potřebuje pro svou činnost. Tím, co předchází vzniku tohoto problému, je v drtivé většině případů nevhodné použití priorit vláken (4.1.2) a to, že jedno vlákno (díky své vysoké prioritě) má neustále přiděleny systémové prostředky a ostatní vlákna s nižší prioritou musí čekat na jeho ukončení. Řešením je nepoužívat předefinování priority vláken a všem ponechat výchozí hodnotu `Thread.NORM_PRIORITY`.

6.1.2.2 Vzájemné brzdění (Livelock)

K této situaci dojde, pokud jedno vlákno (ačkoliv není zablokováno jako v deadlocku) nevykonává žádnou činnost, protože se opakovaně pokouší vykonat operaci, která neustále selhává. Tato situace může nastat například při instant messagingu nebo v aplikaci typu klient-server (popřípadě peer-to-peer), kdy se změní stav subjektu přijímacího zprávu a odesilatel se pokouší neustále tuto zprávu odeslat. Většina těchto programů je koncipována tak, že zprávy k odeslání se řadí do fronty a pokud odeslání selže, vrací se na začátek fronty.

Další možností je situace analogická tomu, kdy se dva lidé potkají proti sobě na chodbě a každý uhýbá na stejnou stranu a jsou tak neustále proti sobě. V aplikaci tato situace může nastat například tehdy, pokud dva klienti odešlou požadavek na server přesně ve stejný okamžik. Dojde k selhání, klienti počkají např. vteřinu a odešlou paket znovu. Opět ale ve stejný okamžik.

Řešením je zabudování náhodnosti do našich programů. Pokud řešíme selhání požadavku opakováním za nějakou jednotku času, je vhodné tuto jednotku času generovat náhodně. Tím minimalizujeme (i když úplně neodstraňujeme) situaci, kdy na sebe dva požadavky narážejí ve stejnou dobu a kolidují.

6.2 Zlepšování výkonu

Jedním z hlavních důvodů, proč používat vlákna, je zlepšení výkonu. Aplikace jsou pružnější, můžeme řídit spouštění podúkolů. Nicméně samotné použití vláken a využívání tříd, které Java nabízí, nám nezaručí výkonnou aplikaci. Zvyšování výkonu za pomoci užití vláken tedy spíše než využití těchto tříd znamená **testování, analýza problému a bezproblémový návrh**. Jak jsme viděli, Java nabízí obrovské množství prostředků usnadňujících návrh vícevláknových aplikací. Nejsme tedy limitováni jazykem, ale spíše svými schopnostmi. Některé použité techniky mohou být kontraproduktivní a dvojsečné, „vylepšení“ jedné části může drasticky snížit výkon jiné. Je nutné od začátku navrhovat program s ohledem na efektivitu, úprava hotového programu může být časově náročnější než tvorba nového, lépe navrženého. Pokud je aplikace rychlá a efektivní, je možné ji ještě doladit a výkon o něco zvýšit, nicméně špatně napsaný a navržený program bude pomalejší, i kdyby programátor uměl čarovat.

Pokud přemýšlíme o výkonu aplikace, musíme brát v potaz několik věcí. Tou první je hardware, na kterém náš program poběží. Výkon je vždy limitován tím, na čem naše aplikace poběží, aneb řetěz je vždy tak silný, jak silný je jeho nejslabší článek. Toto platí nejen v souvislosti s hardwarem, ale i vzhledem k návrhu programu – jedno slabé místo zpomalí celou aplikaci. Vícevláknové aplikace jsou náročnější na hardware, vytváření a správa vláken jsou operace, vyžadující větší režii, zejména přepínání kontextu (viz. kapitola 2.2.4) . Proto je hodně zcestná myšlenka, že čím více vláken použijeme, tím lepší náš program bude. Tou nejzásadnější myšlenkou je co nejefektivněji

využít procesor a systémové prostředky. Neznamená to na 100% vytížit CPU, ale chytře využít výpočetní výkon a zbytečně nedimenzovat požadavky.

6.2.1 Filosofie výkonných aplikací

Nejjednodušší cesta, jak zjistit výkonnost aplikace, je změření výkonu. Neexistuje univerzální nástroj, který by nám prověřil aplikaci a vrátil nějaké číslo, které bude vyjadřovat výkon našeho programu. Pro kompletní testování a ověření je třeba hodnotit z více pohledů a zaměřit se na více aspektů. Můžeme měřit část běhu, propustnost dat, dobu odezvy, účinnost, využití systémových prostředků. Měření výkonu vícevláknových aplikací se musí ubírat úplně jinou cestou než u tradičních jednovláknových programů. U těch se sledovala efektivita programu např. pouze podle času běhu a složitosti algoritmu. U vícevláknových aplikací se zaměřujeme na rozdělení problému a co nejefektivnější paralelismus celého programu. Rychlost a složitost jsou dvě věci, které mohou být vedle sebe odděleně, na rozdíl od jednovláknových programů. Nevyplývá z toho ovšem, že by to nebylo důležité. Naopak. Jde pouze o to vhodně odhadnout, kdy a jak použít více vláken a správně navrhnout aplikační logiku. Pak **můžeme** získat aplikaci s vyšším výkonem.

Důležitou zásadou, která se často zmiňuje, je neuspěchat optimalizace. Správný program je takový, který funguje správně, teprve potom můžeme dělat optimalizace. Důležitá je volba správného algoritmu, efektivita se ladí později.

Hodně věcí ovlivní také rozhodnutí, kolik vláken použít. Někdy je výhodné rozdělit úkol mezi hodně vláken, pokud se jedná o časově náročnou úlohu s konstantní rychlostí. Stejně tak použití více procesorů dokáže urychlit aplikaci. Rozhodně ale neplatí, že přidáním druhého procesoru se program zrychlí dvakrát. Brian Goetz uvádí, že program, vykonávající z 90% paralelní úkoly, bude zrychlen na deseti procesorech zhruba pětkrát, na sto procesorech devětkrát. Vytížení procesoru prudce klesá zhruba k počtu

dvaceti procesorů, pak už je jen lehce klesající, tedy program poběží zhruba stejně na stroji s dvaceti procesory jako na stroji se stem.

Důležitým místem a možnou slabinou našich programů může být i doba, po kterou je držen zámek. Nehledě na to, jak výkonný máme systém, se tak tato doba může stát tím, co zásadně ovlivní výkon programu. Pokud je zámek držen 2ms, nelze vykonat více, než 500 operací za vteřinu (reálně bude toto číslo nižší). Měli bychom tedy minimalizovat kód uvnitř synchronized bloků a použít jen to nejnutnější. U jednoduchých programů rozdíl nepoznáme, u rozsáhlých aplikací bude pokles výkonu znatelnější. Například může někdy být vhodnější použít pouze synchronizovaný blok kódu namísto celé metody. Náročnou (ale účinnou technikou) je použití většího množství zámků – pokud by měla aplikace pouze jeden zámek a ten si předávala, stane se z ní vlastně serializovaná aplikace a nebude v jejím rámci existovat žádný paralelismus. Pokud použijeme větší množství zámků, za cenu vyšší režie ovšem získáme větší pružnost aplikace a vlákna budou čekat kratší dobu na přidělení zámku.

6.2.2 Monitorování výkonu CPU

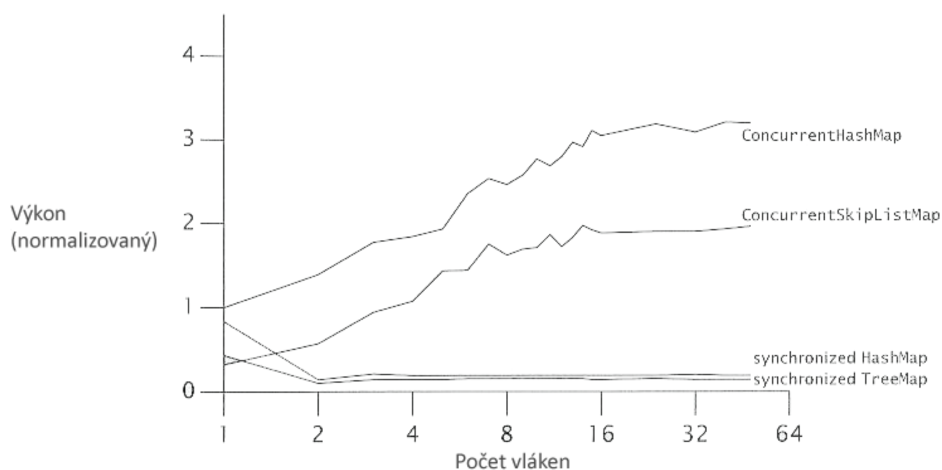
Pokud testujeme aplikaci na výkon, jde nám o to zjistit, jak účinně využívá systémové zdroje. Ještě důležitější je tato část na vícejádrových a víceprocesorových systémech, kdy sledujeme vytížení jednotlivých procesorů a můžeme tak odhalit, jak by bylo ještě možné náš program vylepšit. Pro testování výkonu na Windows můžeme použít například utilitu *perfmon*, která je součástí operačního systému. Pokud zjistíme, že aplikace např. asymetricky využívá procesory (jeden pracuje naplno a ostatní víceméně stojí), je to pro nás důvod pro zvýšení paralelismu aplikace. Toto však bude záležet přímo na konkrétním projektu a systému, na kterém bude nasazen. Pro každý počítač můžeme získat jiné výsledky a z nich vyvodit různá opatření, opravdu záleží na konkrétním nasazení aplikace. Důvody nerovnoměrného vytížení však nemusí být pouze nízký stupeň paralelismu naší aplikace. Může se jednat o záležitost I/O operací, připojení ke vzdálené

databázi nebo např. špatně zvolenou implementaci, či nevhodně použitý objekt.

Na následujícím příkladu si ukážeme, jak moc může ovlivnit výkon výběr vhodné kolekce. V projektu potřebujeme použít „slovník/mapu“ (viz. kapitola 5.5.3) ve vícevláknovém prostředí. Java nám nabízí několik implementací rozhraní `Map`, jak v knihovně `java.util.concurrent` tak v `java.util`. Pro použití ve vícevláknové aplikaci můžeme buď použít `HashMap` a `TreeMap` v kombinaci se `synchronized` blokem a nebo sáhnout do knihovny `java.util.concurrent`, kde najdeme `ConcurrentHashMap` a `ConcurrentSkipListMap`.

Náš příklad bude založen na jednoduché operaci náhodného výběru klíče a vrácení hodnoty, pokud nebude klíč nalezen, bude hodnota s určitou pravděpodobností přidána, pokud bude nalezen, bude hodnota s menší pravděpodobností odstraněna. Tyto operace budou spouštěny ve stoupajícím počtu vláken, a budeme sledovat výkon, přepočítaný na normalizovanou stupnici.

Zjistíme, že s narůstajícím počtem vláken přistupujících k mapě stoupá i výkon `ConcurrentHashMap` a `ConcurrentSkipListMap`, zatímco po přidání druhého vlákna klesne u synchronized map téměř na nulu. V případě jednoho vlákna je výkon v podstatě srovnatelný, obrovské rozdíly přicházejí až po nasazení do vícevláknového prostředí.



Obr. 6 – porovnání výkonu map

6.2.3 Shrnutí

Jedním z nejzásadnějších důvodů, proč používat vlákna, je využití výkonu více procesorů. Ve vícevláknových aplikacích má největší vliv na výkon poměr kódu, který je spouštěn paralelně, k serializovanému kódu. Vhodnou prací se zámky, objekty a propracovaným návrhem aplikace tak můžeme získat vysoký výkon i přehledný a účinný kód.

6.3 Testování vícevláknových programů

Jakkoliv jsou vícevláknové aplikace výkonnější, jsou také náchylnější k chybám. S každým vláknem navíc se zvyšuje pravděpodobnost interakce a chyby, je proto třeba věnovat velkou pozornost návrhu a plánování. Paralelní aplikace můžeme testovat na výkon stejně jako sekvenční, navíc si ale musíme ohlídat problémy, které vyplývají z vícevláknovosti. Zaměřit se musíme zejména na dvě věci. Tou první je vláknová bezpečnost a konzistence dat, tou druhou pak předcházení problémům s aktivitou.

6.3.1 Testování na správnost

Vývoj testovacích jednotek pro vícevláknové aplikace je odvozen od vývoje pro klasické jednovláknové aplikace. Za prvé oddělit invarianty a podmínky, které se dají zkontrolovat mechanicky. Za druhé, napsat testy pro kód, který mechanicky otestovat nelze. Pro testování lze v Javě využít testovací rámec JUnit [http://junit.org], který umožní kontrolovat správnost výpočtu, vytvářet testovací objekty jako instance naší třídy a dále s nimi pracovat. Umožní testovat výsledky metod a připravit testy tak, že nemusíme do programu zadávat data, ale použijeme už předpřipravený testovací přípravek. Výhodou různých testovacích frameworků je i to, že vytváří různé výpisy a logy, ze kterých snadněji zjistíme, proč naše aplikace selhala. Zároveň je více než vhodné pro testování v prvním stadiu využívat profilery a debuggery, které jsou součástí mnoha vývojových prostředí.

Pokud chceme, aby test opravdu fungoval, měla by být testovací data a výsledky testů neodhadnutelná kompilátorem. Je více než nevhodné používat pořád stejná data pro každý běh testu. Chyba v algoritmu může být taková, že se projeví pouze pro některá data. Takové chyby jsou více než časté – hranice intervalu podmínky, kladná / záporná čísla, ošetření operací s nulou, špatně inicializované objekty a práce s nimi, přetečení rozsahu, … Takových příkladů je mnoho. Proto by se testovací přípravky měly zaměřit právě na tyto chyby a odhalit je dříve, než bude program předváděn klientovi a chyba se poprvé objeví právě tam.

6.3.2 Testování na výkon

Kromě hraničních a nulových hodnot bychom měli program testovat s pseudonáhodnými čísly (popř. texty či jinými daty). Je vhodné napsat si vlastní metodu, generující náhodná čísla, a nespoléhat na hotové metody z API, náhodnost u nich totiž není příliš velká. Vhodnou eventualitou se ukazuje například kód generující náhodná čísla z kombinace systémového

času a hashování, popř. ještě bitového posunu. Tato metoda nám umožní vracet náhodná čísla s mnohem větším rozptylem, nízkou četností a různorodostí pro každý běh programu.

Stejně tak by naše programy měly být testovány na dostatečně velkém vzorku dat, pět hodnot není dostatečné množství pro opravdové otestování správnosti. Výsledky testů budou o to lepší, pokud donutíme program pracovat až na samé hranici kapacity, čímž otestujeme zároveň i jeho efektivitu a ošetření práce se zdroji. Zároveň by programy měly být otestovány jak na slabých strojích, tak na strojích s vysokým výkonem. Správně napsaný vícevláknový program by měl na obě situace reagovat pružně a nemělo by docházet k velkým nárůstům či poklesům výkonu.

6.3.3 Profilery a monitory

Velká vývojová prostředí nabízejí nástroje pro podporu práce s vlákny. Zprostředkovávají podrobný pohled na to, jak vlastně náš program pracuje, spousta z nich může i ovlivnit běh vlákna (časování). Jsou vytvořeny tak, aby nám co nejvíce zjednodušily situaci a zpřehlednili program. Zobrazují např. různými barvami vlákna v různých stavech, stejně tak jako podrobné grafy a monitory využití výkonu počítače. Pokročilejší profilery umí například i odhalit, kde jsou slabá místa programu a kde by mohlo dojít k problémům s aktivitou.

Jednoduchou možnost analýzy vláken nám nabízí i knihovní třída `java.lang.management.ThreadInfo`, poskytující informace nejen o spuštění vlákna, ale i jednoduché statistiky synchronizace.

6.3.4 Nejčastější chyby vícevláknových programů

V této kapitole zmíním nejčastější chyby, se kterými se programátor může potkat při psaní vícevláknových aplikací. Jedná se jak o chyby sémantické, tak logické. Jedná se víceméně o vzory (patterns) těch nejběžnějších chyb, se kterými se může programátor potkat.

- **Spouštění Thread.run()**
  Vlákno implementující rozhraní `Runnable` musí mít implementovánu metodu `run()`. Spouštění vlákna však probíhá přes volání metody `Thread.start()`.

- **Neuvolnění zámku**
  Explicitní zámky nejsou uvolňovány samy a o jejich ukončení se musí postarat programátor. Vhodné je například uvolnění zámku v bloku `finally`.

- **Spouštění vlákna z konstruktoru**
  Při použití dědičnosti může způsobit problémy.

- **Nadbytečná notifikace vláken**
  Zavoláním `notify()` (nebo `notifyAll()`) říkáme, že se objekt změnil. Měly by být volány pouze v případě změny objektu, ne jen jako součást `synchronized` bloku. V některých příkladech jsou tyto metody volány „pro jistotu“, i když objekt nebyl změněn.

- **Chybné použití Condition**
  Volání metod wait() nebo await() by mělo probíhat ve smyčce. Volání těchto metod bez zámku, mimo cyklus nebo bez otestování stavu podmínky končí téměř vždy chybou.

7 Závěr

V této diplomové práci jsem se pokusil shrnout možnosti, jaké nabízí Java pro tvorbu vícevláknových aplikací. Java od verze Java 5 nabízí pokročilé možnosti pro tyto programy, v tomto ohledu se jedná o silnou vlastnost jazyka. Java 5 přinesla nejen přepracovaný paměťový model, ale také novou knihovnu `java.util.concurrent`, která zahrnuje objekty a rozhraní připravené na míru nejčastějším problémům a situacím, se kterými se lze setkat při psaní vícevláknových aplikací.

Práci jsem pojal jako komplexní pohled na tvorbu vícevláknových aplikací, zaměřil jsem se i na aspekty, které by měl čtenář znát, pokud chce pracovat s vlákny v Javě. Pro pochopení problémů, které mohou nastat, je vhodné seznámit se s technologickým pozadím Javy, zejména s tím, jak Java pracuje s pamětí a z čeho vznikají problémy se synchronizací a interakcí vláken. Do této skupiny zapadá i teorie operačních systémů, správy paměti, plánování a algoritmů spojených s touto činností. Teorie multitaskingu a toho, jak operační systém spravuje jednotlivé procesy, je tou nejvhodnější analogií pro vlákna. Mezi filosofií multiprocesových systémů a vícevláknových programů lze nalézt velké množství styčných bodů.

Tuto práci jsem rozdělil v podstatě na čtyři logické celky. Tím prvním je teoretický úvod, kde se zabývám nejen principy multitaskingu, ale také lehce nahlédnu na technologii Java. Druhou částí je úvod do práce s vlákny, kde se věnuji objektům typu `Thread` a základním principům vláken. Třetí částí je pak popis knihovny `java.util.concurrent`, zdůraznění důležitých tříd a rozhraní. Tyto tři části dávají dohromady teoretický soubor potřebných znalostí pro psaní vícevláknových programů. Nicméně, jak jsem několikrát v této práci zdůraznil, tvorba vícevláknových aplikací je obtížná a má řadu úskalí, se kterými se můžeme potkávat. Některá z nich byla zmíněna již v předchozích částech, zbytek se nachází v části čtvrté, věnované testování a výkonu. Snažil jsem se zde vytvořit jakýsi soubor tipů, který má ukázat, jakým směrem by se

měl ubírat vývoj a hlavně návrh aplikací. Analýzu a návrh považuji za důležitější, než samotné psaní kódu. Vzhledem k rozsahu práce jsem nemohl řadu věcí rozepsat více, neboť tato práce má více sloužit jako vstup do světa vícevláknových aplikací než být vyčerpávající příručkou.

Vývoj vláknových aplikací v Javě tedy nespočívá jen v nastudování knihovních tříd a rozhraní, ale zejména v pochopení síly, kterou využití vláken nabízí. Použití vláken v našich programech může výrazným způsobem zvýšit výkon, knihovna `java.util.concurrent` nabízí silné prostředky pro podporu paralelismu aplikací. Druhou stranou mince jsou problémy, které použití vláken přináší - interference vláken, konzistence paměti, problémy s aktivitou. Největší chybou by bylo myslet si, že použití vláken automaticky náš program vylepší. Pro vhodné případy získáme nárůst výkonu i zpřehlednění návrhu, při špatném použití program nemusí fungovat vůbec.

Javu považuji za velmi dobře navržený jazyk, a prostředky, které nabízí pro práci s vlákny, jsou toho důkazem. Pokud bude mít programátor na paměti všechny zákeřnosti, která vlákna mohou způsobit a dokáže navrhovat programy efektivně a v souladu se zásadami uvedenými v této práci, pak dokáže tvořit výkonné a moderní aplikace.

Seznam přiložených projektů

Tato diplomová práce obsahuje několik ukázkových příkladů, které doprovázejí samotný text. Všechny jsou přiloženy na CD ve formě projektů NetBeans a BlueJ. Projekty jsou rozděleny na dvě hlavní části – projekty doprovázející text diplomové práce a hlavní projekty, demonstrující použití vláken. Jména projektů doprovázejících text diplomové práce jsou uvozeny číslem, evokujícím kapitolu, ke které se vztahují.

Projekty doprovázející text diplomové práce

412_-_priorita-vlaken

Projekt demonstruje použití priority vláken a to, jak ovlivní střídání vláken.

421_-_vytvoreni-vlakna-pomoci-runnable

Vytvoření vlákna pomocí implementace rozhraní `Runnable`.

421_-_tridici-algoritmy

Demonstrace spuštění několika vláken, v každém vláknu běží jeden řadící algoritmus pro seřazení velkého pole čísel.

422_-_vytvoreni-vlakna-pomoci-dedicnosti

Vytvoření vlákna oddědĕním od třídy `Thread`.

430_-_pozastaveni-behu-vlakna

Program ukazuje, jak lze jednoduše pomocí metody `sleep()` pozastavit běh vlákna.

44_-_zaklady-prace-s-vlakny

Projekt je shrnutím základů práce s vlákny a metodami, používanými při práci s vlákny.

451_-_interference-vlaken

Na projektu jednoduchého čítače je ukázán případ, kdy není zaručen správný výsledek programu, díky interferenci vláken a nerovnoměrnému střídání vláken.

464_-_volatile-promenne

Šablona, demonstrující použití `volatile` proměnné typu `boolean` pro zajištění správné funkce programu ve vícevláknovém prostředí.

523_-_cyklicka-bariera

Ukázka použití bariéry jako synchronizačního primitiva.

54_-_prace-s-executorem

Jednoduchá ukázka práce s objektem typu `Executor` a semaforem, jakožto implementovaným synchronizačním primitivem.

544_-_scheduled-executor-service

Ukázka odsunutého spuštění vlákna, vlákno je spuštěno deset vteřin po spuštění programu.

5442_-_prace-s-executor-service

Třída ukazuje jednoduchou práci s ExecutorService a skupinami vláken (Thread Pool)

551_-_producent-konzument

Šablona demonstruje použití vláken a implementace fronty pro simulaci klasické úlohy typu producent/konzument

553_-_mapa

Práce s objektem typu `ConcurrentMap`.

Projekty demonstrující použití vláken

AnimaceGIF

Projekt ukazuje, jak pomocí vláken a skupiny statických obrázků vytvořit animaci na principu animovaného GIFu.

AnimaceNezavisla

Spuštění animace několika objektů, které se pohybují nezávisle na sobě.

BallGame

Hra, kde je úkolem hráče trefovat míček, který se náhodně vystřeluje z horní části obrazovky.

GUI

Tlačítko Storno, které umožní ukončení dlouhé činnosti, která by jinak způsobila zamrznutí programu.

GUI2

16 barevných čtverečků, které náhodně mění barvu.

SimpleChat

Jednoduchá aplikace typu klient-server, použitelná pro výměnu textových zpráv.

Klíčová slova

Vývoj vícevláknových aplikací

Programování

Java 5

java.util.concurrent

Paralelní programy

Multitasking

Multithreading

Seznam použitých zdrojů

[1] GOETZ, Brian. *Java Concurrency in Practice*. 3rd edition. USA : AddisonWesley, 2006. 403 s. ISBN 0-321-34960-1.

[2] Sharon Zakhour a kolektiv. *Java 6 - Výukový kurz*. 1. vyd. Brno : Computer Press a.s., 2007. 534 s. ISBN 978-80-251-1575-6.

[3] ECKEL, Bruce. *Thinking in Java*. 4th edition. USA : Prentice Hall, 2006. 1482 s. ISBN 0-13-187248-6.

[4] HEROUT, Pavel. *Učebnice jazyka Java*. 1. vyd. České Budějovice : Kopp, 2001. 349 s. ISBN 80-7232-115-3.

[5] LEWIS, Bil, BERG, Daniel J. *Multithreaded Programming with Java Technology*. USA : Sun Microsystems Inc., 2000. 457 s. ISBN 0-13-017007-0.

[6] HORTON, Ivor. *Java 5*. Praha : Neocortex, 2005. 1443 s. ISBN 80-86330-12-5.

[7] NEJEDLÝ, Petr. *Paralelní programování v Javě a java.util.concurrency : Videozáznam přednášky na ČVUT*. 2006. Dostupný z WWW: <http://www.avc-cvut.cz/avc.php?id=3513>.

[8] PICHLÍK, Roman. *Synchronizace, JMM a další špeky - díl první* [online].
2002-2008 , 17.6.2007 [cit. 2007-10-25]. Dostupný z WWW: <http://www.sweb.cz/pichlik/archive/2007_06_17_archive.html#7856042966845756553>.

[9] PICHLÍK, Roman. *Synchronizace, JMM a další špeky - díl druhý* [online]. 2002-2008 , 20.6.2007 [cit. 2007-10-25]. Dostupný z WWW:

<http://www.sweb.cz/pichlik/archive/2007_06_17_archive.html#5962449848552968299>.

[10] GOETZ, Brian. *JSR 133 in Public Review* [online]. 1995-2007 , 13.4.2004 [cit. 2007-10-03]. Dostupný z WWW: <http://today.java.net/pub/a/today/2004/04/13/JSR133.html>.

[11] GOETZ, Brian, MANSON, Jeremy. *JSR 133 (Java Memory Model) FAQ* [online]. 2004 , February 2004 [cit. 2007-10-12]. Dostupný z WWW: <http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html>.

[12] *DeveloperWorks : Java technology* [online]. [2007] [cit. 2007-10-30]. Dostupný z WWW: <http://www.ibm.com/developerworks/java/>.

[13] GOETZ, Brian. *Brian Goetz: Publications* [online]. 2000- [cit. 2007-09-20]. Dostupný z WWW: <http://www.briangoetz.com/pubs.html>.

[14] *Java 6 SE Documentation* [online]. Dostupný z WWW: <http://java.sun.com/javase/6/docs/api/index.html>.