<embed/it>

Vlákna v Javě

PV168
29. 3. 2016
Petr Adámek

Obsah

- Co jsou to vlákna
- Jak vlákna fungují
- Vytváření vláken v Javě
- Paměť a vlákna
- Synchronizace přístupu k paměti
- Java Memory Model
- Další možnosti synchronizace

Co jsou to vlákna

Co jsou to vlákna

- Mechanismus, který umožňuje souběžné vykonávání více činností (čili posloupností operací) najednou.
- Jedno vlákno reprezentuje právě jednu posloupnost operací, která je prováděna v rámci nějakého procesu.
- Každý proces musí mít minimálně jedno vlákno, ale může jich mít více.

Příklady použití vláken

- **Desktopové aplikace**
  - Jedno vlákno provádi časově náročnou operaci (např. prohledávání disku)
  - Jiné vlákno obsluhuje uživatelské rozhraní
- **Webové aplikace**
  - Několik souběžných vláken obsluhuje několik souběžných HTTP požadavků
- **Výpočetně náročné aplikace**
  - Např. AFIS (biometrický server) používá pro prohledávání databáze otisků prstů tolik vláken, kolik je v systému k dispozici CPU jader, aby se maximáně využil výpočetní výkon systému a doba trvání operace se minimalizovala

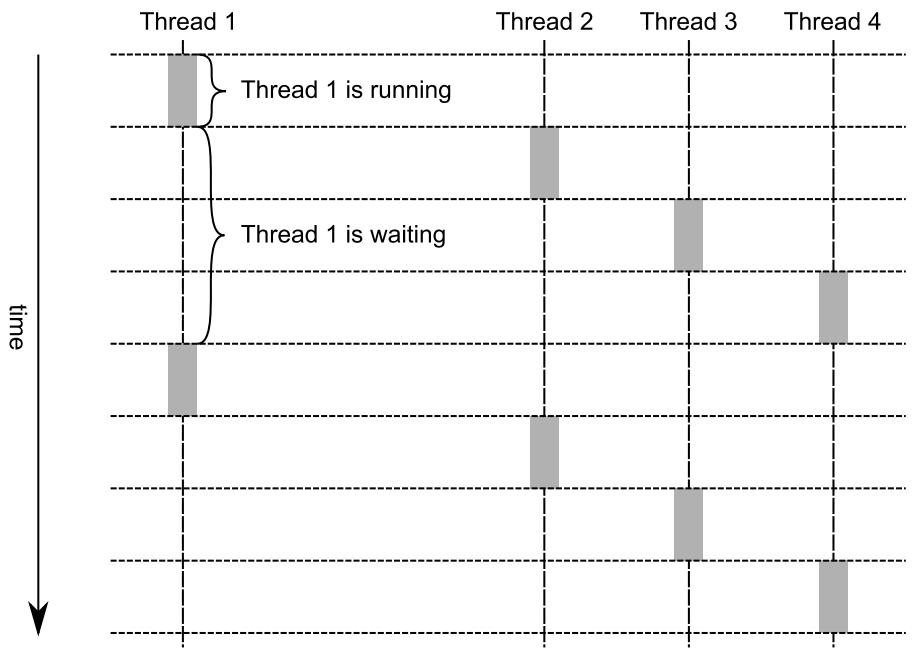
Vlákna v Javě

- **Drtivá většina našich aplikací bude vícevláknových**
  - Desktopové aplikace ve Swingu (Swing je navržen tak, že používá více vláken)
  - Webové aplikace a podnikové informační systémy (souběžné požadavky jsou obsluhovány různými vlákny)
- **Java obsahuje dobrou podporu pro tvorbu vícevláknových aplikací, nicméně je s nimi spojena řada záludností, které nemusí být na první pohled vůbec zřejmé**
  - Je třeba problematice vláken dobře rozumět
  - Chyby v synchronizaci vláken se projeví pouze za určitých okolností

Jak vlákna fungují

Jak vlákna fungují

- **Jednoprocesorové systémy**
  - Pouze jedna posloupnost operací (jedno vlákno)
  - Iluze souběžného běhu je dosahována *přepínáním kontextu*
  - Preemptivní multitasking (přepínání kontextu probíhá bez kooperace s vláknem)
  - K přepínání kontextu dochází mnohokrát za vteřinu

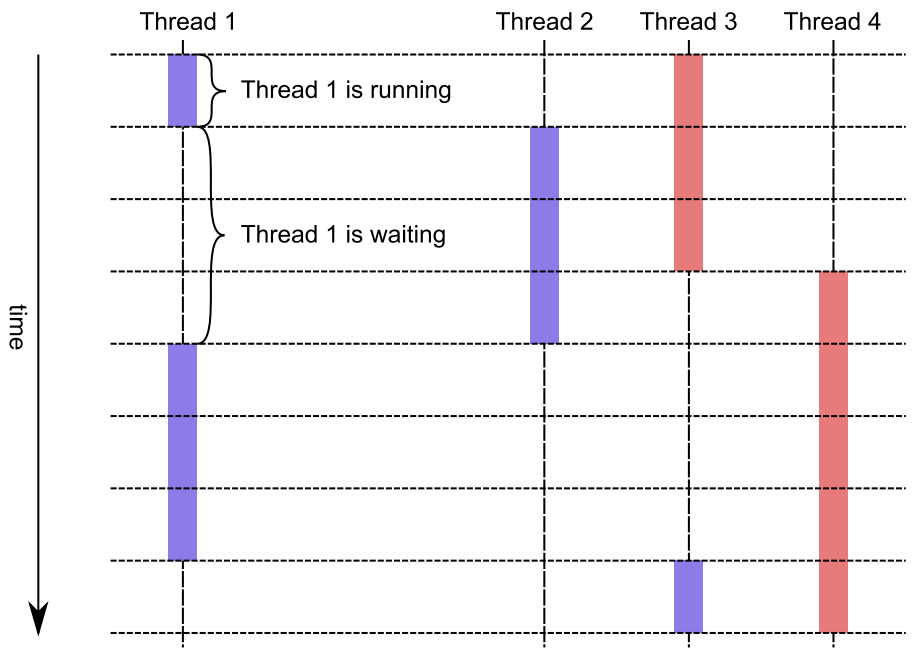


Víceprocesorové systémy

- **Dnes je standardem**
  - Více CPU v jednom systému
  - Více jader v jednom CPU
  - Více souběžných vláken v rámci jednoho jádra (zejména u RISC)
- **SPARC T5 (1024 souběžně běžících vláken)**
  - Až 8 CPU v jednom systému
  - 16 jader v jednom CPU
  - 8 souběžných vláken v jednom jádře
- **Intel Xeon E5-2695v2 (48 souběžně běžících vláken)**
  - Až 2 CPU
  - 12 jader v jednom CPU
  - 2 vlákna v jednom jádře (hyperthreading)

Jak vlákna fungují

- **Víceprocesorové systémy**
  - Více vláken skutečně běžících souběžně
  - Stále dochází k *přepínání kontextu*
  - Plánovač vláken se obvykle snaží spouštět jedno vlákno na stejném CPU, ale není to garantované



Přepnutí kontextu

- **K přepnutí kontextu dochází, když vlákno**
  - spořebuje přidělený čas;
  - požádá o uspání na určitý počet milisekund (`Thread.sleep(long)`);
  - čeká na dokončení blokující operace (např. `InputStream.read()` či `reader.read()` pokud nejsou data připravena a je nutné na ně čekat);
  - chce vstoupit do kritické sekce hlídané monitorem v níž se již nachází jiné vlákno (čili pokud chce vykonat kód, který nesmí být vykonáván více vlákny souběžně a je momentálně zamčený, protože jej vykonává jiné vlákno);
  - využívá operaci, která mu umožňuje čekat na jiné vlákno (tj. zavolá metodu `Object.wait()` nebo `Thread.join()`);
  - navrhne, že mu může být procesor odebrán (`Thread.yield()`) – pozor, plánovač vláken to může ignorovat a s vyjímkou speciálních případů je lepší se této metodě vyhnout (viz kontrakt metody yield).

Vytváření vláken v Javě

Vytváření vláken Javě

- **Nové vlákno můžeme vytvořit a spustit**
  - Rozšířením třídy **`Thread`** a překrytím metody **`run()`**. Pak vytvoříme instanci této třídy a zavoláme metodu **`Thread.start()`**. Toto je zastaralý a nedoporučený způsob.
  - Vytvořením třídy (nebo lambda výrazu) implementující rozhraní **`Runnable`**. Instanci této třídy pak předáme jako parametr konstruktoru třídy Thread a zavoláme metodu **`Thread.start()`**.
  - Vytvořením třídy (nebo lambda výrazu) implementující rozhraní **`Runnable`**. Instanci této třídy pak předáme jako parametr metody **`execute`** nějaké vhodné implementace rozhraní **`Executor`**. Tento způsob poskytuje užitečnou abstrakci oddělující požadavek na provedení dané úlohy od konkrétního mechanismu provedení. Není tak např. nutné pro každou úlohu vytvářet nové vlákno, ale je možné vlákna recyklovat nebo používat techniku *thread pooling*.

Vytváření vláken Javě – rozšíření třídy Thread

```
class CounterThread extends Thread {
    // Tato metoda obsahuje kód, který bude vykonáván v našem vlákně
    public void run() {
        for(int i = 0; i < 10; i++) {
            System.out.println(i);
        }
    }
}

// Vytvoříme nové vlákno
Thread counterThread = new CounterThread();

// spustíme vlákno, kód metody CounterThread.run() se od této chvíle
// začne vykonávat v novém vlákně
counterThread.start();
```

Vytváření vláken Javě – Runnable & Thread

```
// Tento lambda výraz obsahuje kód,
// který bude vykonáván v našem vlákně
Runnable counter = () -> {
        for(int i = 0; i < 10; i++) {
            System.out.println(i);
        }
    };

// Vytvoříme nové vlákno, jako parametr konstruktoru předáme
// referenci na naši implementaci rozhraní Runnable
Thread counterThread = new Thread(counter);

// spustíme vlákno, kód výše uvedeného lambda výrazu se od této chvíle
// začne vykonávat v novém vlákně
counterThread.start();
```

Vytváření vláken Javě – Runnable & Executor

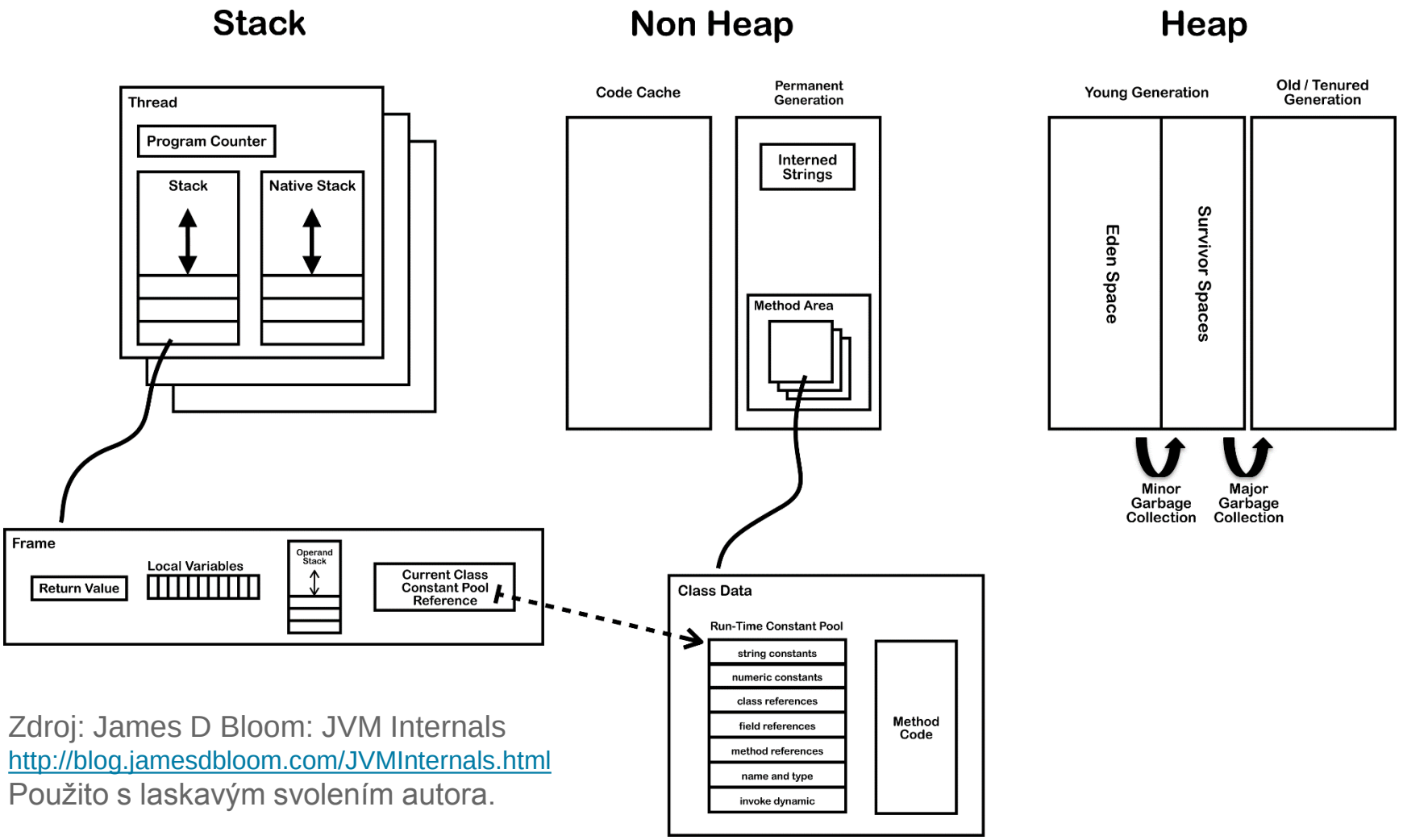
```
// Tento lambda výraz obsahuje kód,
// který bude vykonáván v našem vlákně
Runnable counter = () -> {
        for(int i = 0; i < 10; i++) {
            System.out.println(i);
        }
    };

// Vytvoříme executor, který bude recyklovat použitá vlákna
// (nemá samozřejmě smysl vytvářet nový executor pro každou úlohu,
// v reálném programu vytvoříme jeden na začátku, který pak budeme
// používat pro všechny úlohy)
Executor executor = Executors. newCachedThreadPool();

// náš lambda výraz odešleme ke spuštění
executor.execute(counter);
```

Paměť a vlákna

Paměť a vlákna



Zdroj: James D Bloom: JVM Internals
http://blog.jamesdbloom.com/JVMInternals.html
Použito s laskavým svolením autora.

Paměť a vlákna

- **Zásobník (*Stack*) – každé vlákno má vlastní**
  - Lokální proměnné
  - Zásobní operandů
- **Halda (*Heap*) – sdílené mezi vlákny**
  - Prostor, kde se alokují instance objektů a polí
- ***Non-Heap* – sdílené mezi vlákny**
  - Konstanty, zavedené třídy, kód metod, cache JIT kompilátoru

Viz James D Bloom: JVM Internals
http://blog.jamesdbloom.com/JVMInternals.html

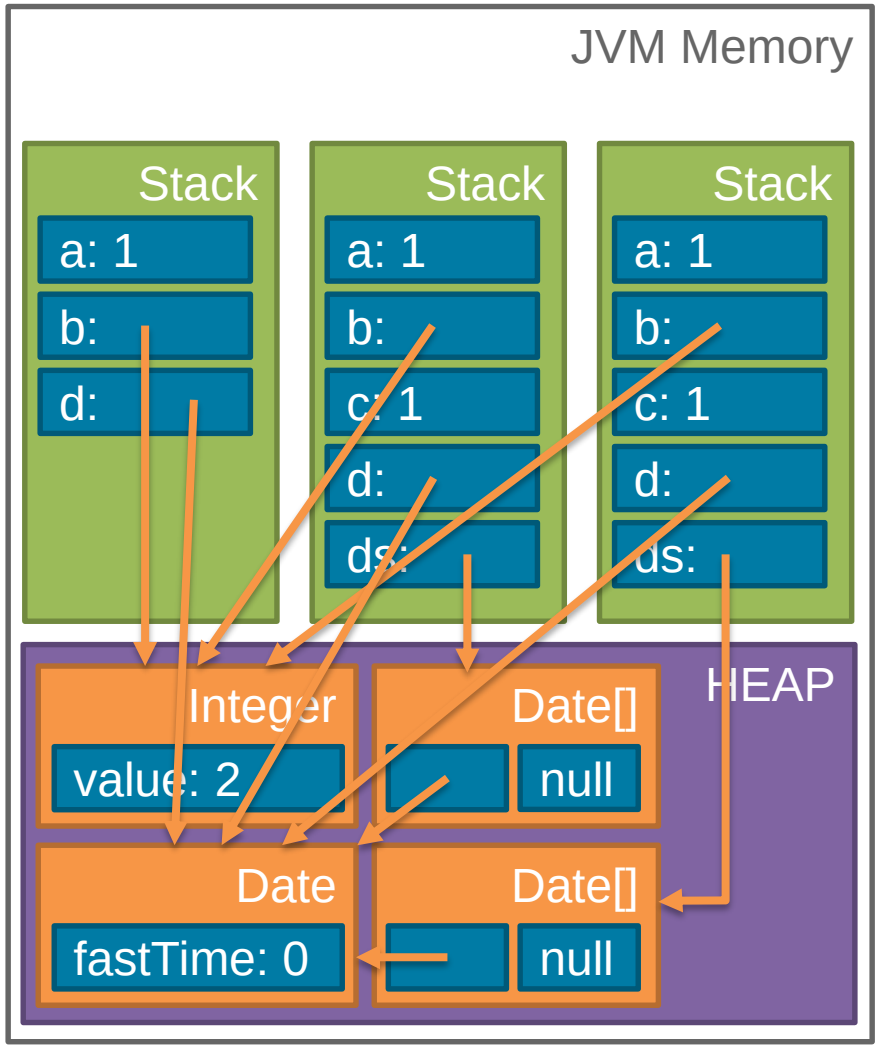
Primitivní a objektové typy

- **Primitivní typy**
  - byte, short, int, long, double, float, char, boolean
  - obsahují přímo hodnotu (tj. parametry se předávají hodnotou)
- **Objektové typy**
  - pole a objekty
  - obsahují odkaz na heap (tj. parametry se předávají odkazem)
- **Neměnitelné třídy (immutable classes)**
  - Jako objektové typy se předávají se odkazem
  - Díky neměnitelnosti se ale chovají stejně, jako by byly předávány hodnotou
  - Musí být ale opravdu neměnitelné! (viz Item 15 v Effective Java)

Paměť a vlákna

```
public class Example {
  public static void main(
      String[] args) {
    int a = 1;
    Integer b = 2;
    Date d = new Date(0);
    new Thread(() -> m(a,b,d))
      .start();
    new Thread(() -> m(a,b,d))
      .start();
  }

  public static void m(int a,
      Integer b, Date d) {
    int c = a;
    Date[] ds = new Date[]
      { d, null };
  }
}
```



Synchronizace přístupu k paměti

Synchronizace přístupu k paměti

Při přístupu ke sdílenému obsahu paměti z různých vláken je nutné zajistit synchronizaci, abychom předešli tzv. Race Condition.

Pozor! Chyby v synchronizace jsou velice zákeřné a obtížně odhalitelné. Obvykle se nedají reprodukovat ani odhalit testováním. Jejich důsledky však mohou být fatální. Proto je nutné správné synchronizace věnovat zvýšenou pozornost a je potřeba znát všechny související aspekty. Některé věci v této oblasti jsou velmi neintuitivní (jsou ve skutečnosti jinak, než by se mohlo na první pohled zdát).

Proč je nutná synchronizace přístupu k paměti?

- **Pokud hovoříme o synchronizaci přístupu k paměti, pak synchronizace plní dva důležité úkoly:**
  - **Brání vláknu v tom, aby pozorovalo nějaký objekt v nekonzistentním stavu.** Pokud nějaké vlákno zrovna mění stav objektu, ostatní vlákna nemohou jeho stav číst nebo měnit, dokud není změna dokončena. Každá změna stavu objektu se tak ostatním vláknům jeví jako atomická.
  - **Pokud jedno vlákno dokončí změnu stavu objektu, ostatní vlákna okamžitě vidí nový stav.** Jednotlivé změny stavu objektu tak na sebe navazují a tvoří posloupnost s určitým pořadím. Nemůže dojít ke dvěma souběžným změnám stavu objektu, vždy nejdříve proběhne jedna a teprve po ní další.

Zejména na druhý bod se často zapomíná, což může vést k fatálním následkům!

Příklad: neatomický zápis hodnoty long

Předpokládejme, že na 32bitovém systému potřebujeme změnit hodnotu typu long z 0 (0x0000000000000000) na na -1 (0xFFFFFFFFFFFFFFFF). Operace nemůže být provedena atomicky, proto se nejdříve změní spodních 32 bitů a poté horních 32 bitů:

1) `0x0000000000000000 → 0x00000000FFFFFFFF`

2) `0x00000000FFFFFFFF → 0xFFFFFFFFFFFFFFFF`

Pokud bude jiné vlákno číst hodnotu po provedení prvního kroku, ale před provedením druhého kroku, přečte hodnotu 4 294 967 295 (0x00000000FFFFFFFF).

Monitor

- **Základním synchronizačním prostředkem v Javě je tzv. Monitor**
- **Monitor umožňuje**
  - Zajistit vzájemné vyloučení (dvě vlákna nemohou provádět stejnou kritickou sekci naráz).
  - Zajistit, aby vlákno vidělo aktuální stav sdíleného objektu (viz *Java Memory Model*).
  - Provést efektivní časovou synchronizaci vláken (jedno vlákno čeká na výsledek operace prováděného druhým vláknem).
- **Každý objekt (tj. každá instance) má svůj vlastní monitor.**

Synchronizace pomocí monitoru

- **Jak synchronizovat**
  - Identifikujeme kritickou sekci (tj. úsek kódu, kde potřebujeme zajistit vzájemné vyloučení)
  - Kritickou sekci označíme klíčovým slovem **`synchronized`**.
- **Klíčovým slovem `synchronized` můžeme označit**
  - celou metodu; pak se u nestatické metody použije monitor příslušné instance (tj. this), u statické metody monitor příslušné třídy (tj. class);
  - blok kódu; použije se monitor objektu, který je uvedený jako parametr v závorce.

Synchronizace pomocí monitoru

```
class Counter {
    // sdílená proměnná reprezentující stav objektu
     private int currentValue = 0;
    public synchronized int next() {
        // toto je kritická sekce, která musí proběhnout atomicky
        return ++currentValue;
    }
}
```

```
class Counter {
    // sdílená proměnná reprezentující stav objektu
    private int currentValue = 0;
    public int next() {
        synchronized(this) {
            // toto je kritická sekce, která musí proběhnout atomicky
            return ++currentValue;
        }
    }
}
```

Atomické operace

- **Java garantuje atomičnost operací čtení a zápisu pro hodnoty typu**
  - byte, short, int, char, float, boolean
  - Reference na objektový typ
- **Atomičnost není garantovaná pro čtená a zápis hodnot typu**
  - long, double
- **Pokud chceme zajistit, aby ostatní vlákna okamžitě viděla provedené změny, atribut musí být označený klíčovým slovem `volatile`.**
- **Lepší je používat třídy z balíku java.util.concurrent.atomic**

Java Memory Model

Java Memory Model

- **Jak se může stát, že jedno vlákno nevidí okamžitě změnu stavu objektu, provedenou jiným vláknem?**
- **Za to může *Java Memory Model***
  - JSR 133 (https://jcp.org/en/jsr/detail?id=133), součást Java 5.
  - Popisuje, jakým způsobem vlákna pracují s pamětí a jak se vzájemně ovlivňují.
  - Umožňuje JVM provádět efektivní optimalizaci, aniž by to mělo dopad na správné fungování programu.
  - Vznikl jako reakce na chyby a nedostatky původního memory modelu v Java 1.4 a starších.
  - První pokus o detailní definici paměťového modelu pro některý z populárních jazyků, ostatní jazyky (např. C++) následovaly později
  - https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html

Optimalizace

- **JVM může provádět řadu optimalizací pro zrychlení běhu programu**
  - Uložení proměnné do registru
  - Změna pořadí prováděných operací
  - Optimalizace pro NUMA architekturu

Příklady optimalizací: proměnná v registru

Operace	Kód bez optimalizace	Doba trvání	Kód s optimalizací	Doba trvání
int a = 0;	mov [a], 0	60 ns (zápis DRAM)	mov eax, 0	1 ns
a = a + 5;	add [a], 5	2 ns (čtení L1 cache) 1 ns 60 ns (zápis DRAM)	add eax, 5	1 ns
int b = a;	mov [b], [a]	2 ns (čtení L1 cache) 1 ns 60 ns (zápis DRAM)	mov [b], eax	60 ns (zápis DRAM)
Celkem:		**186 ns**		**62 ns**

Čísla jsou pro Core i7 Xeon 5500 a jsou pouze přibližná. Zdroj: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

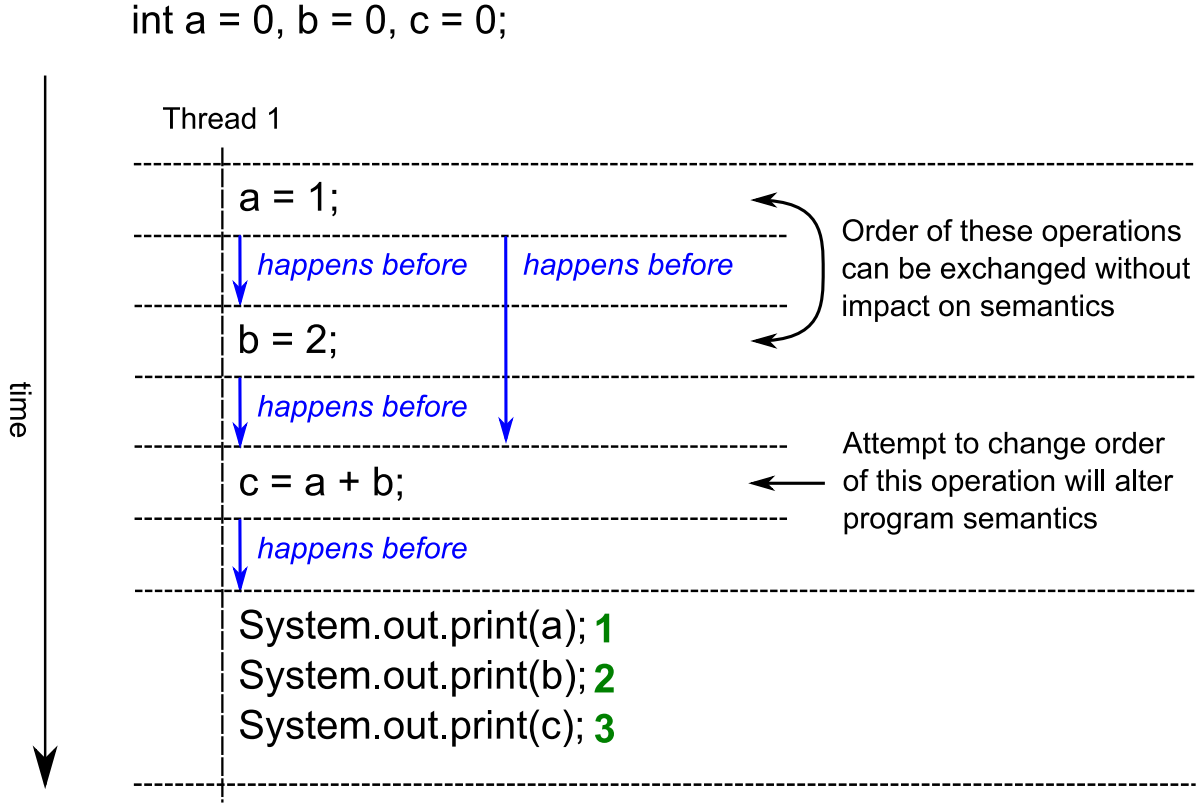
Tato optimalizace přinesla trojnásobné zrychlení, ale hodnota proměnné *a* není nikdy zapsána do paměti, takže ostatní vlákna ji nikdy neuvidí.

Základní princip: *Happens Before*

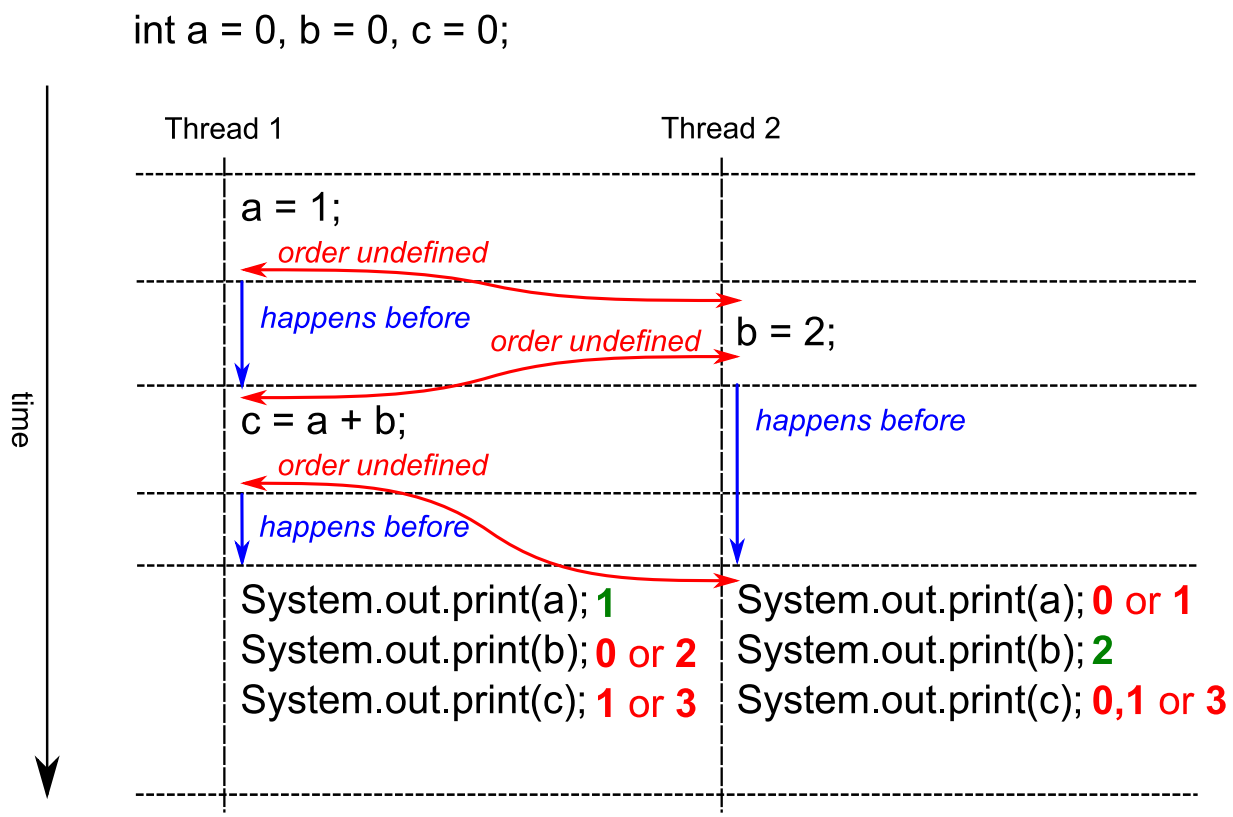
- ***Java Memory Model* definuje relaci částečného uspořádání pro operace s pamětí (*read field*, *write field*, *lock*, *unlock*) a s vláknem (*start*, *join*).**
  - Each action in a thread *happens before* every action in that thread that comes later in the program's order.
  - An unlock on a monitor *happens before* every subsequent lock on that same monitor.
  - A write to a volatile field *happens before* every subsequent read of that same volatile.
  - A call to start() on a thread *happens before* any actions in the started thread.
  - All actions in a thread *happen before* any other thread successfully returns from a join() on that thread.

Happens Before – Single thread

Each action in a thread happens before every action in that thread that comes later in the program's order.



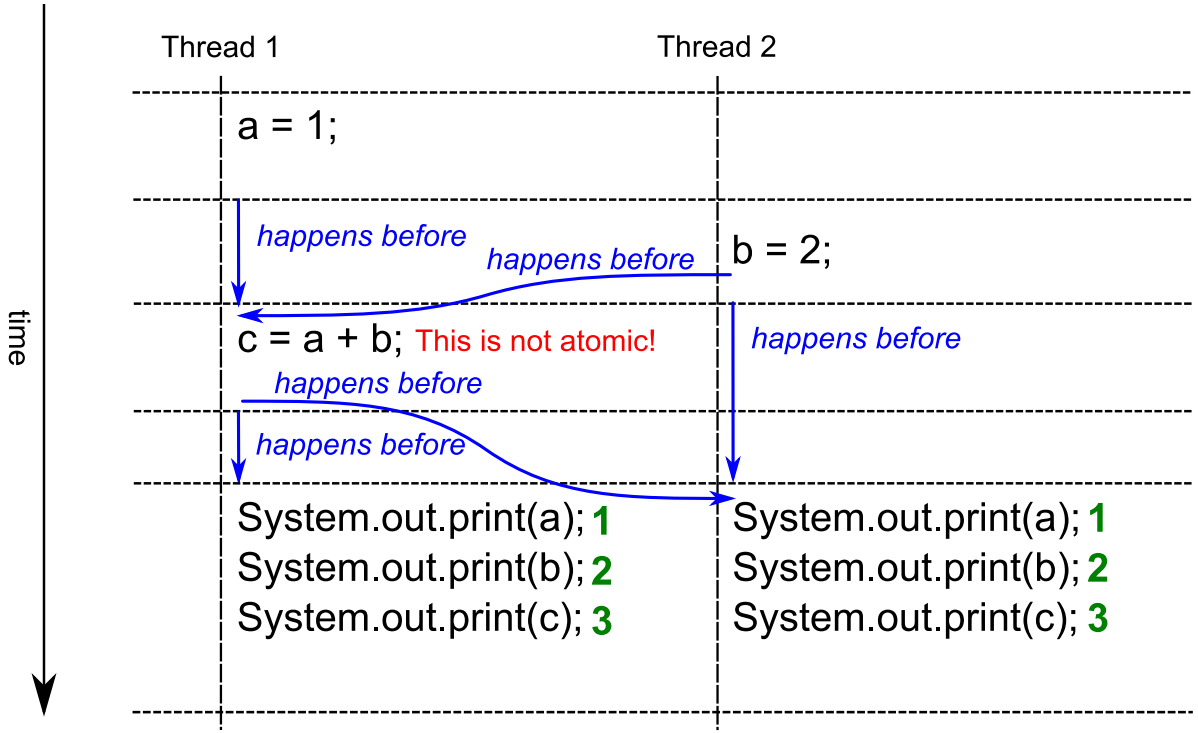
Happens Before – No synchronization



Happens Before – Volatile

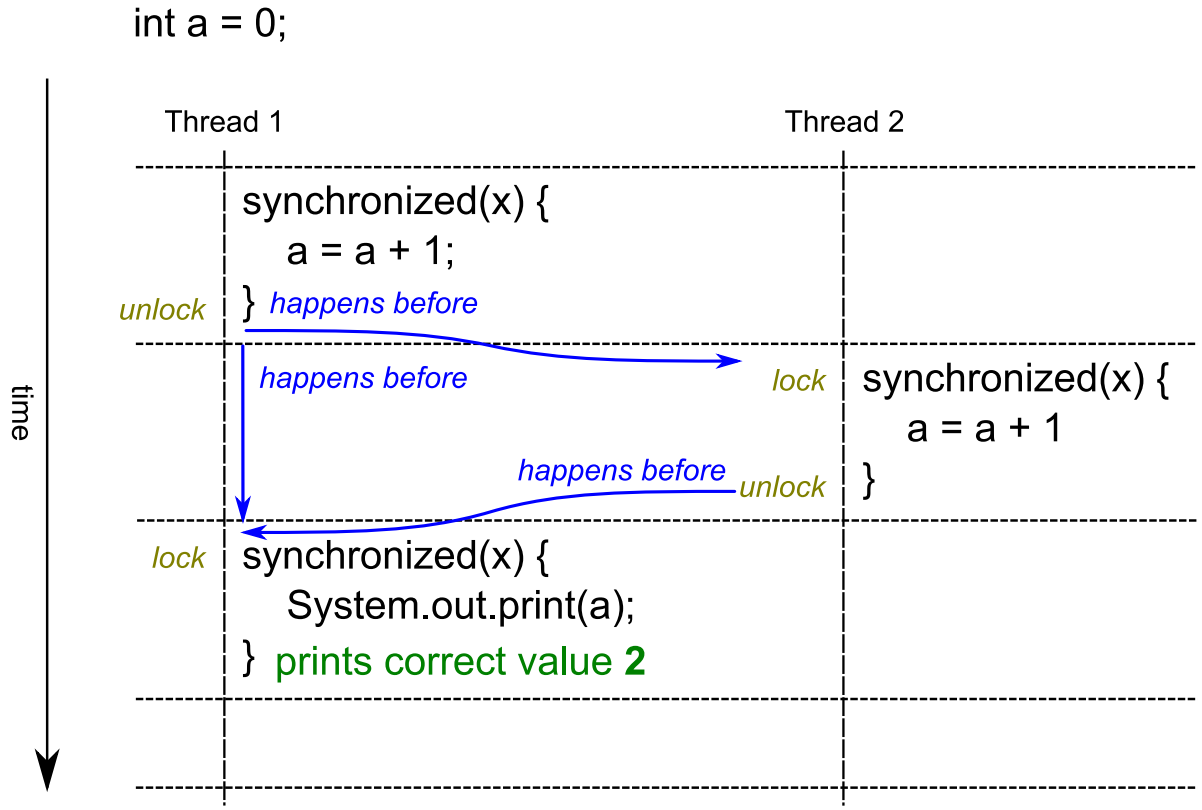
A write to a volatile field happens before every subsequent read of that same volatile.

volatile int a = 0, b = 0, c = 0;



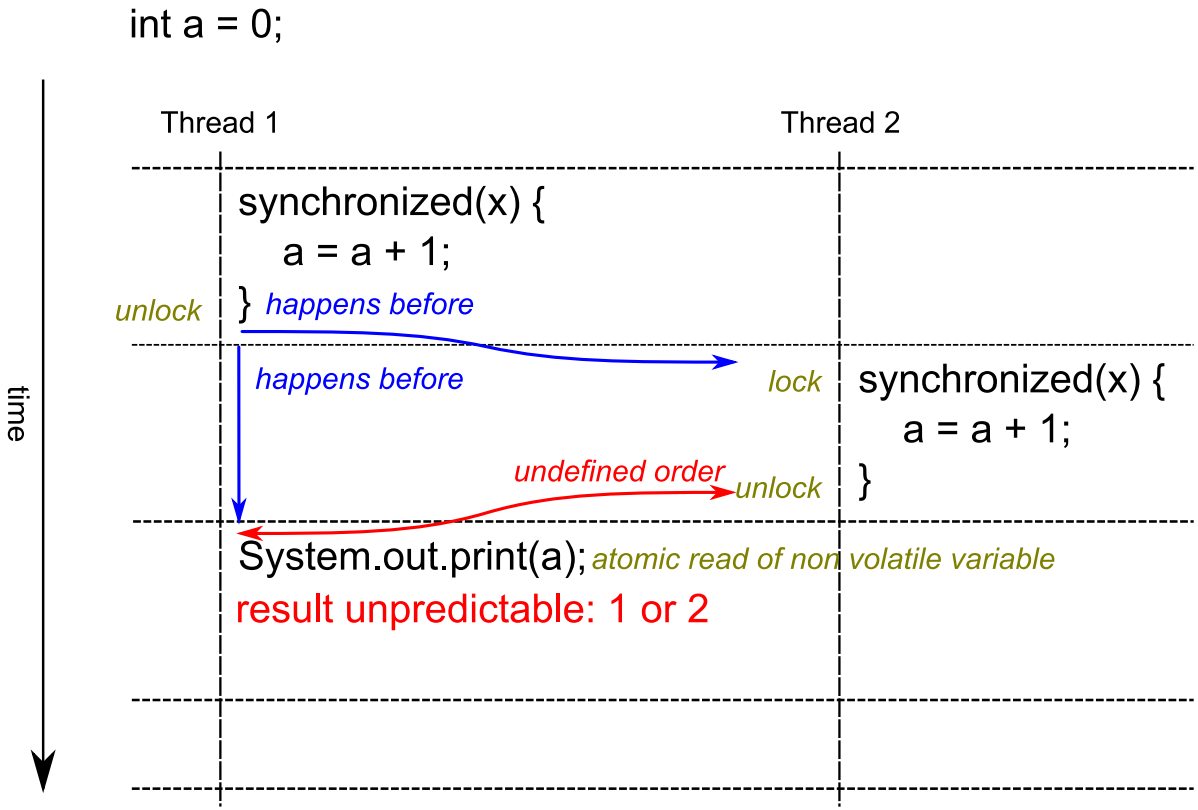
Happens Before - Synchronized

An unlock on a monitor *happens before* every subsequent lock on that same monitor.



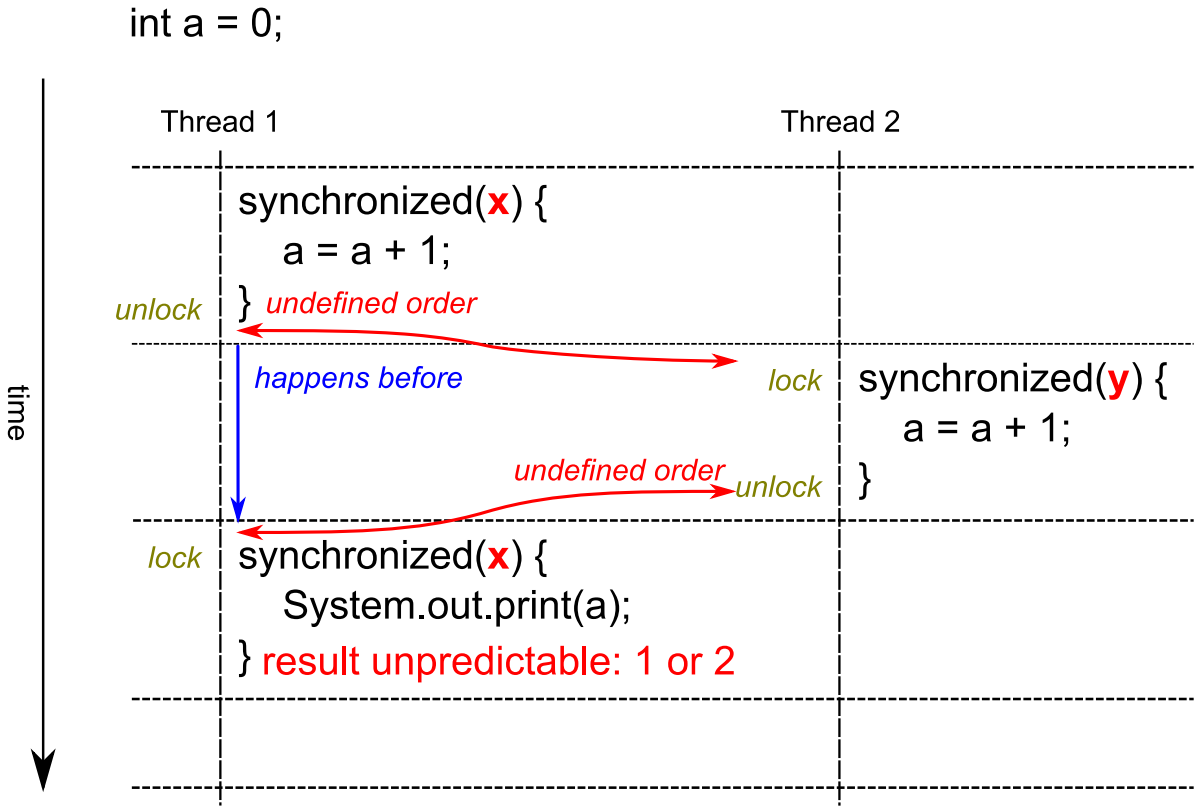
Missing read synchronization

An unlock on a monitor *happens before* every **subsequent lock** on that same monitor.



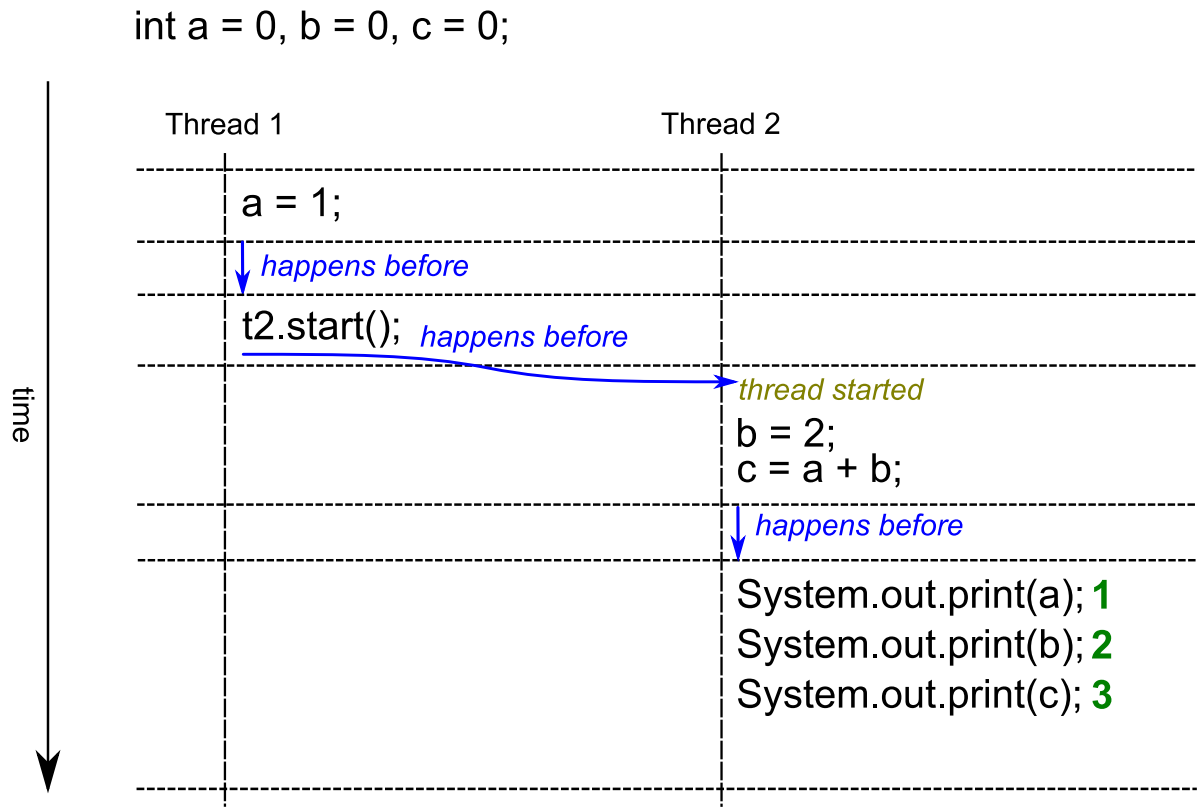
Not same monitor

An unlock on a monitor *happens before* every subsequent lock on that **same** monitor.



Happens Before – start()

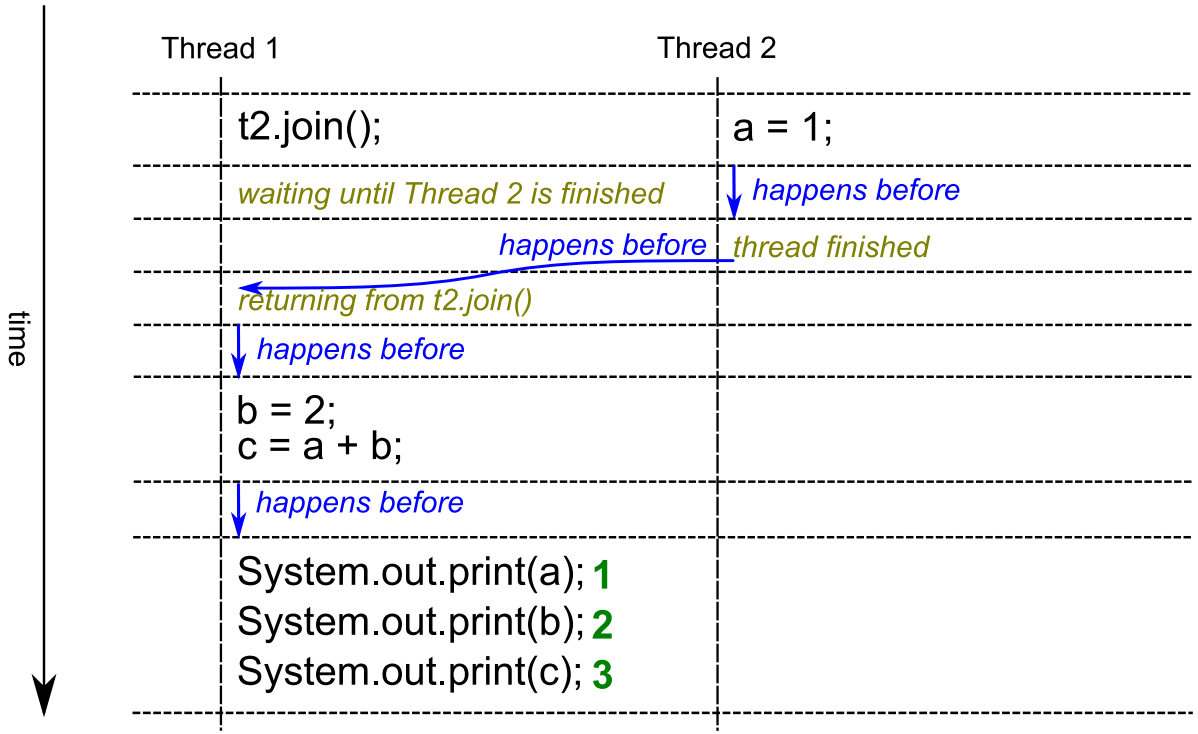
A call to start() on a thread happens before any actions in the started thread.



Happens Before – join()

All actions in a thread happen before any other thread successfully returns from a join() on that thread.

```
int a = 0, b = 0, c = 0;
```



Kdy není třeba synchronizovat

Kdy není třeba synchronizovat

- **Synchronizace není třeba**
  - Při přístupu k proměnným a objektům, k nimž se vůbec nepřistupuje z jiných vláken (nejsou tudíž sdílené).
  - Při přístupu k instanci třídy, která je vláknově bezpečná (tj. je sama synchronizovaná).
  - Při přístupu ke finálním proměnným, pokud není reference na this předčasně zpřístupněna ostatním objektům (pokud používáme javu verze 5 nebo novější).
  - Při přístupu k lokálním proměnným. Ty jsou uloženy na zásobníku vláka a nejsou tedy sdíleny mezi více vlákny. Ale pozor, lokální proměnná může obsahovat referenci na objekt na heapu, který už může být sdílený. Nemusíme tedy synchronizovat manipulaci s hodnotou lokální proměnné, ale pro manipulaci s odkazovaným objektem už to platit nemusí.
  - Při atomických operacích nad proměnnými označenými klíčovým slovem volatile (pokud používáme javu verze 5 nebo novější, podrobněji viz dále).

Další možnosti synchronizace

Synchronizace pomocí Lock

```
public class Counter {
    // sdílená proměnná reprezentující stav objektu
    private int currentValue = 0;
    // zámek
    private final Lock lock = new ReentrantLock();

    public int next() {
        lock.lock();
        try {
            // toto je kritická sekce, která musí proběhnout atomicky
            return currentValue++;
        } finally {
            lock.unlock(); // Unlock musí být vždy ve finally bloku!!!
        }
    }
}
```

java.util.concurrent.locks.Lock

- **Výhody**
  - Menší riziko nesprávného použitítryLock()
  - Možnost nastavení timeoutu
  - Přerušitelné operace
  - Čekání na podmínku
  - Fair ordering policy

Časová synchronizace vláken

- **Řízené pomocí metod**
  - Object.wait()
  - Object.notify()
  - Object.notifyAll()
- **Pozor na časté chyby, lepší je použít třídy z balíku java.util.concurrent nebo java.util.concurrent.locks**
- **viz Item 69 v Effective java**

Užitečné třídy v Java Core API

- **java.util.concurrent.atomic**
  - AtomicInteger, AtomicBoolean, AtomicLong
- **java.util.concurrent.locks**
  - Lock, Condition, ReadWriteLock
- **java.util.concurrent**
  - CopyOnWriteArrayList, ConcurrentHashMap, BlockingQueue, Executor, Executors, Semaphor, CountDownLatch, CyclicBarrier
- **Join-fork framework**
  - Paralelní streamy

Další operace s vlánky

Zastavení a pozastavení vlákna

- **Pokud chceme zastavit vlákno, je třeba jeho aktivní spolupráce**
  - Definujeme vláknově bezpečný příznak (např. `AtomicBoolean`), jehož nastavením můžeme signalizovat požadavek na ukončení vlákna.
  - Vlákno pravidelně stav tohoto příznaku kontroluje a když zjistí, že byl nastaven na hodnotu **`true`**, samo se ukončí.
  - Můžeme využít i mechanismus *přerušení* (viz následující slajd).
- **Podobně implementujeme i pozastavení vlákna**
- **Nepoužívejte deprecated metody `Thread.stop()`, `Thread.suspend()` a `Thread.resume()`!**
  - Viz Java Thread Primitive Deprecation.

Zastavení vlákna

```
// příznak požadavku na zastavení vlákna
private final AtomicBoolean stop = new AtomicBoolean();


// Create and start the thread
new Thread(() -> {

    while(!stop.get()) {
        // do your task
    }

}).start();


// Stop the thread
stop.set(true);
```

Přerušení vlákna

- **Vlákno může být v případě potřeby přerušeno voláním metody `Thread.interrupt()`**
  - Pokud vlákno čeká v blokující operaci, která podporuje mechanismus přerušení (např. wait(), sleep(long) or join()), operace je přerušena a příslušná metoda vyhodí `InterruptedException`.
  - V opačném případě je vláknu nastavený příznak *interrupted*. Pokud mechanismus chceme využívat, vlákno by mělo stav tohoto příznaku pravidelně kontrolovat a adekvátně na něj ragovat.
- **Zjištění stavu příznaku interrupted**
  - `Thread.isInterrupted()` – vrátí stav příznaku *interrupted* daného vlákna.
  - `Thread.interrupted()` – vrátí stav příznaku *interrupted* aktuálního vlákna a pokud je nastaven, provede jeho vymazání.

Ukončení aplikace

- **Aplikace je ukončena**
  - Pokud zavoláme metodu `System.exit(int)`.
  - Pokud je ukončeno poslední vlákno (s výjimkou vláken, které mají nastavený příznak *deamon*).

Vliv vláken a synchronizace na výkon

Synchronizace a vliv na výkon

- **Synchronizace může mít významný vliv na výkon**
  - Samotná synchronizace má nezanedbatelnou režii.
  - Synchronizace znemožňuje provádět řadu optimalizací.
  - Kromě toho může mít další vliv na výkon, pokud je synchronizace nevhodně navržená, nebo implementovaná.
- **Nepoužívejte synchronizaci, pokud to není nutné**
  - Původně byly všechny třídy v Java Core API synchronizované.
  - Java 6 někdy umí nadbytečnou synchronizaci eliminovat.
- **I když je synchronizace nutná, dávejte si pozor, ať ji navrhnete dobře.**

Rizika chyb v synchronizaci

- Strádání
- Uvíznutí

Počet vláken a jejich priorita

- **Udržujte počet vláken v aplikaci na rozumně nízkém počtu**
  - Plánovač vláken se na každé platformě může chovat jinak a je navíc ovlivněn tolika faktory, že je pro nás jeho chování prakticky nedeterministické (viz Effective Java, Item ??). Čím více je v aplikaci vláken, tím je rozhodování plánovače složitější a roste riziko, že je bude plánovat jinak, než bychom potřebovali.
  - *Thread pooling*, `ThreadPoolExecutor`, `SwingWorker`.
- **Nespoléhejte na prioritu vláken**
  - `Thread.setPriority(int)`
  - Interpretace priorit je silně závislá na konkrétní platformě a proto je nutné postupovat při práci s prioritami vláken velmi obezřetně.
  - Mnohem účinnější je počet vláken omezit

Pár poznámek na závěr

Standarní synchronizační úlohy

- **Consumer-Producer**
  - BlockingQueue
- **Readers-writers**
  - ReadWriteLock
- **Dining philosophers**

Rekapitulace důležitých zásad

Zdroje

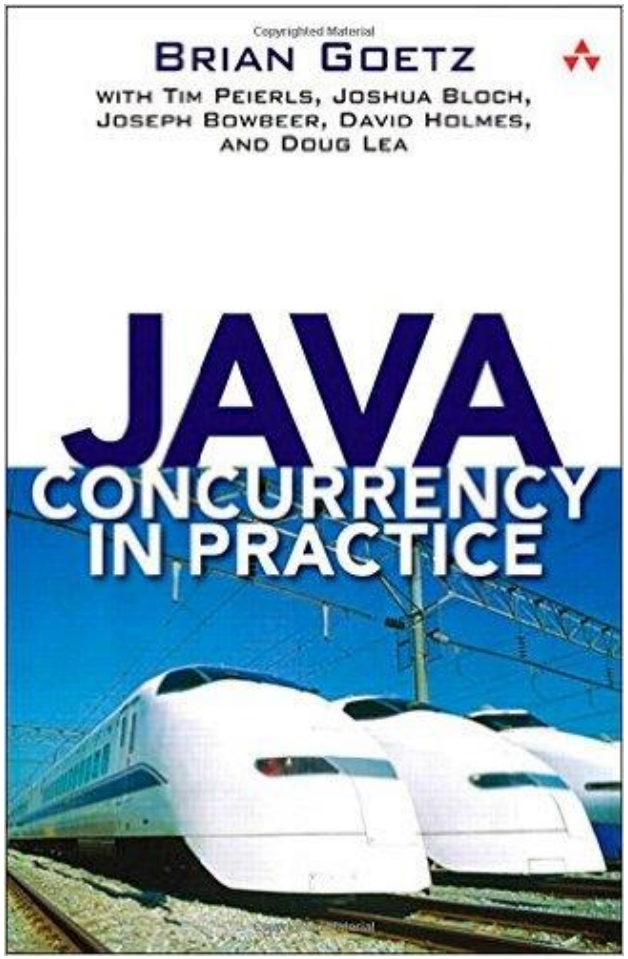
Effective Java (2nd Edition)

Joshua Bloch

http://amazon.com/dp/0321356683/

(Threads are covered with Items 66 – 73)

Zdroje

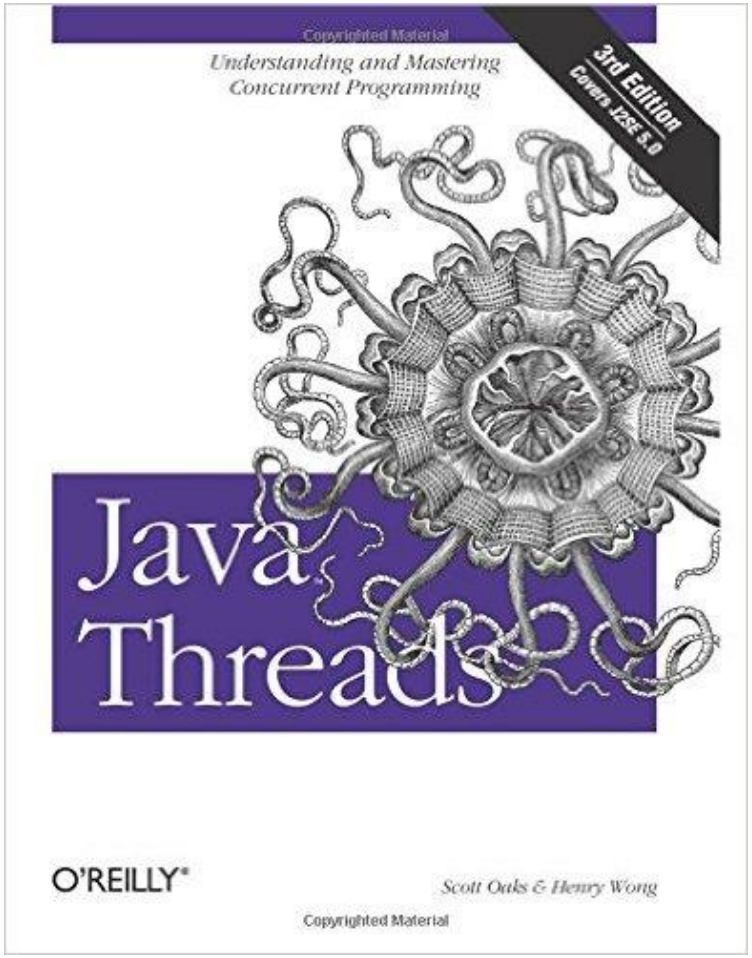


Java Concurrency in Practice

Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea

http://amazon.com/dp/0321349601/

Zdroje



Java Threads (3nd Edition)

Scott Oaks, Henry Wong

http://amazon.com/dp/0596007825/

Závěr