Návrh multithreadových aplikací

Synchronizace přístupu ke sdíleným modifikovatelným datům

Zopakování některých pojmů:

- Mutable x immutable
- String, StringBuffer, StringBuilder
- volatile
- Kritická sekce
- Atomické čtení-zápis dat (int, long, double)
- `synchronized`
- monitor

Synchronizace přístupu ke sdíleným modifikovatelným datům

```
class BankovniTransakce {

private BankovniUcet buOdkud;
private BankovniUcet buKam;

...

  synchronized public prevod(double castka)
    buOdkud.odecti(castka);
    buKam.pricti(castka);
  }

...

}
```

Synchronizace přístupu ke sdíleným modifikovatelným datům

- `synchronized` blok se často považuje za konstrukt sloužící k implementaci kritické sekce
  - Na začátku `synchronized` bloku je stav chráněného objektu konzistentní
  - Při opuštění `synchronized` bloku je stav chráněného objektu konzistentní
  - Uvnitř `synchronized` bloku nemusí být stav chráněného objektu konzistentní
- ale to, že implementuje kritickou sekci není všechno!
  – viz dále

Synchronizace přístupu ke sdíleným modifikovatelným datům

- **Nebezpečná rada:**
  - V zájmu vyššího výkonu se vyhýbej synchronizaci a místo toho používej atomické čtení/zápis dat

Synchronizace přístupu ke sdíleným modifikovatelným datům

```
private static int nextSerialNumber = 0;

public static int generateSerialNumber() {
  return nextSerialNumber++;
}
```

Je metoda generateSerialNumber threadově bezpečná?

Synchronizace přístupu ke sdíleným modifikovatelným datům

```
private static int nextSerialNumber = 0;

public static int generateSerialNumber() {
  return nextSerialNumber++;
}
```

Je metoda generateSerialNumber threadově bezpečná?

Není: Operace `nextSerialNumber++` není atomická !!!
Důsledek: Dva různé thready mohou vygenerovat totéž ID

Synchronizace přístupu ke sdíleným modifikovatelným datům

Náprava:

```
private static int nextSerialNumber = 0;

synchronized
public static int generateSerialNumber() {
  return nextSerialNumber++;
}
```

Synchronizace přístupu ke sdíleným modifikovatelným datům

```
public
class StoppableThread extends Thread {
  private boolean stopRequested = false;

  public void run() {
    boolean done = false;
    while (!stopRequested && !done) {
      ... // delej uzitecny vypocet
    }

  public void requestStop() {
    stopRequested = true;
}
```

Boolská proměnná stopRequested je atomicky čtena / modifikována. Je to threadově bezpečné?

Synchronizace přístupu ke sdíleným modifikovatelným datům

```
public
class StoppableThread extends Thread {
  private boolean stopRequested = false;
```

Boolská proměnná stopRequested je atomicky čtena / modifikována. Je to threadově bezpečné?

Není: proměnná `stopRequested` nemusí být po každém updatu ukládána do paměti – může být držena v registru.

Důsledek: Když je v threadu T1 zavolána metoda `requestStop()`, thread T2 nemusí zaznamenat změnu proměnné `stopRequested`.

Synchronizace přístupu ke sdíleným modifikovatelným datům

Náprava:

```
public
class StoppableThread extends Thread {
  private boolean stopRequested = false;

  public void run() {
    boolean done = false;
    while (!stopRequested() && !done) {
      ... // delej uzitecny vypocet
    }

  public
  synchronized boolean stopReqquested() {
    return stopRequested;
}

  public synchronized void requestStop() {
    stopRequested = true;
}
```

Synchronizace přístupu ke sdíleným modifikovatelným datům

Náprava:

```
  public
  synchronized boolean stopReqquested() {
    return stopRequested;
}

  public synchronized void requestStop() {
    stopRequested = true;
}
```

`synchronized` je zde potřeba přesto, že se nejedná o kritickou sekci (není třeba zamezit současnému vykonání metody více thready, protože operace jsou atomické)!

Synchronizace přístupu ke sdíleným modifikovatelným datům

Jiná možnost nápravy:

```
public
class StoppableThread extends Thread {
  private boolean volatile stopRequested = false;

  public void run() {
    boolean done = false;
    while (!stopRequested && !done) {
      ... // delej uzitecny vypocet
    }

  public void requestStop() {
    stopRequested = true;
}
```

Synchronizace přístupu ke sdíleným modifikovatelným datům

Lazy inicializace:

```
private static Foo foo = null;

public static Foo getFoo() {
  if (foo == null) {
    synchronized (Foo.class) {
      if (foo == null) {
        foo = new Foo();
    }
  }
  return foo;
}
```

Zde zabezpečujeme přístup ke statickým datům.

Úmysl je zřejmý.

- Není třeba zatěžovat režií synchronizace každý přístup k `foo`.
- Synchronizace je potřeba, jenom pokud proměnná `foo` ještě nebyl inicializována.
- proměnnou `foo` testuji ještě jednou (uvnitř kritické sekce), protože po předchozím testu mohla být inicializována jiným threadem.

Je metoda `getFoo()` threadově bezpečná?

Synchronizace přístupu ke sdíleným modifikovatelným datům

Lazy inicializace:

```
synchronized (Foo.class) {
  if (foo == null) {
    foo = new Foo();
}
```

- konstrukce objektu, na který odkazuje proměnná `foo` je atomická. Nejprve se zcela zkonstruuje instance třídy `Foo` a pak se reference na ni uloží do `foo`.
  Zatím v pořádku

- Proměnná foo je tedy buďto null nebo odkazuje na plně zkonstruovaný objekt.
  Zatím v pořádku

- Přesto, když bude jiný thread číst takto zkonstruovaný objekt, nemusí vidět jeho aktuální stav (aktuální hodnoty některých datových členů si zapisující/inicializující thread může držet v registrech). Čtoucí thread tedy může přistoupit k částečně inicializovanému objektu a následné volání některé metody tohoto objektu může skončit katastrofou.
  Zásadní problém

Synchronizace přístupu ke sdíleným modifikovatelným datům

Náprava č. 1:

```
private static Foo foo = new Foo();

public static Foo getFoo() {
  return foo;
}
```

Odstoupili jsme od lazy inicializace, objekt `Foo` vytvoříme hned na začátku, ačkoliv možná vůbec nebude potřeba (metodu `getFoo()` vůbec nikdo nezavolá).

Synchronizační problému odpadnou, ale objekt `Foo` (může být velmi velký) a bude (možná) zbytečně zabírat paměť.

Ta lazy inicializace může mít něco do sebe. Tak jak to udělat, aniž bychom měli „synchronizační“ problémy ?

Synchronizace přístupu ke sdíleným modifikovatelným datům

Náprava č. 2:

```
private static Foo foo = null;

public static synchronized Foo getFoo() {
  if (foo == null)
    foo = new Foo();
  return foo;
}
```

Děláme lazy inicializaci, (potenciálně velký) objekt `Foo` nebude překážet v paměti, když nebude potřeba (metodu `getFoo()` vůbec nikdo nezavolá).

Synchronizační problémy nejsou, metoda `getFoo()` je synchronizovaná.

Zatěžujeme ale každý přístup k `getFoo()` režií synchronizace. Přitom v případech, kdy je foo inicializována, synchronizace není nutná. Na tom původním přístupu bylo něco raciálního. Jak tedy?

Synchronizace přístupu ke sdíleným modifikovatelným datům

Náprava č. 3:

```
private static class FooHolder {
  static final Foo foo = new Foo();
}

public static Foo getFoo() {
  return FooHolder.foo;
}
```

Ačkoliv to nemusí být na první pohled zřejmé, **děláme lazy inicializaci**, (potenciálně velký) objekt `Foo` vznikne při inicializaci statických proměnných třídy Foo, ke kterému **dochází až při prvním použití** (vlastnost jazyka java)**!**

Žádný přístup k `getFoo()` nebude zatížen režií synchronizace.

Synchronizace přístupu ke sdíleným modifikovatelným datům

Shrnutí:

- kdykoliv více threadů sdílí **mutable** data, každý thread, který data **čte** nebo **modifikuje**, musí získat zámek monitoru příslušné `synchronized` sekce.

- Možnou alternativou je kombinace **atomických** operací a **volatile** proměnných, ale
  - je to pokročilá technika – programátor si musí být perfektně vědom toho, co činí
  - při změně modelu paměti v příštích specifikacích javy nemusí fungovat.

Deadlock

Nejen nedostatečná synchronizace ale i přemíra synchronizace může škodit.

Pravidlo:

Nikdy nepředávej řízení klientovi uvnitř synchronizované metody nebo bloku.

Co to znamená?

Uvnitř synchronizované oblasti nevolej `public` nebo `protected` metodu, která je určena k tomu aby byla v některé odvozené třídě `override`n.

Taková metoda je obvykle `abstract` nebo může mít defaultní implementaci.
Z hlediska onoho synchronizovaného regionu je taková metoda **cizincem**. Třída obsahující onen synchronizovaný region nemůže mít žádnou znalost o její budoucí implementaci.

Klient může může poskytnout takovou implementaci tohoto cizince, jež vytvoří nový thread volající zpět metodu třídy obsahující synchronizovaný onen region. Nově vzniklý thread se snaží získat zámek, který ale vlastní ten původní thread.

Oba thready uváznou (deadlock).

```
public abstract class WorkQueue {
   private final List queue = new LinkList();

   protected WorkQueue() {new WorkerThread().start();}

   public final void enqueue(Object workItem) {
      synchronized (queue){
         queue.add(workItem);
         queue,notify();
      }}

   protected abstract void processItem(Object workItem)

   private class WorkerThread extends Thread {
      public void run() {
         while (true) {
            synchronized (queue) {
               while (queu.isEmpty() && !stopped) {
                  queue.wait();
               Object workItem = queue.remove(0);
               processItem(workItem); // volani "cizince"
            }
```

Bude-li cizinec implementován např. takto:

```
class DeadlockQueue extends WorkQueue {

   protected void processItem(final Object workItem) {
      Thread child = new Thread() {
        public child = new Thread() {
          public void run() { enqueue(workItem); }
        }
      }
      child.start();
      child.join(); // Deadlock
   }
}
```

... dojde k **deadlocku**.

Náprava:

Cizinec se musí volat mimo synchronizovaný blok.

```
private class WorkerThread extends Thread {
   public void run() {
      while (true) {
         synchronized (queue) {
            while (queu.isEmpty() && !stopped) {
               queue.wait();
            Object workItem = queue.remove(0);
         }
         processItem(workItem); // volani cizince mimo
                                // synchronizovany region
      }
   }
}
```

Deadlock

Pravidlo:

Cizinec se volá mimo synchronizovaný blok.

je v souladu s obecnějším pravidlem

Pravidlo:

Minimalizuj počet akcí prováděných uvnitř synchronizovaného bloku a prováděj tam jen nezbytné akce.

Deadlock

Pravda:

Volání cizince z vnitřku synchronizovaného bloku je chybou i v případě, že cizinec nespouští další thread. Pokud se by se totiž cizinec obrátil zpět k objektu, z něhož byl zavolán, najde tento objekt potenciálně v nekonzistentním stavu, neboť se Nacházíme uvnitř synchronizovaného bloku.

Proto dříve uvedené pravidlo

Pravidlo:

Cizinec se volá mimo synchronizovaný blok.

má význam nejen jako prevence deadlocku.

Vliv synchronizace na výkon

- Režie spojená se synchronizací výrazně poklesla ve srovnání s původními verzemi javy, ale přesto **stále existuje**.

- Threadově bezpečná třída, která se ale používá většinou v single-threadových v těchto případech zbyutečně zatěžuje procesor režií spojenou se synchronizací. Příklad – třída `StringBuffer` (proto se objevila threadově nezabezpečená třída `StringBuilder`), `BufferedInputStream`.

- Mnohdy je řešením implementovat threadově nezabezpečenou třídu, kterou je možno **externě** synchronizovat:
  - wrapperem (například

```
List aList = Collections.
                    synchronizedList(new ArrayList());
```

  - nebo zděděnou třídou, kdy základní třída není theradově bezpečná, zatímco poděděná třída threadově bezpečná je (vhodné tam, kde se npředpokládá, že kient bude extendovat/reimplementovat základní třídu).

Vliv synchronizace na výkon

- U tříd, u nichž se předpokládá typické nasazení v multithreadovém provozu se preferuje **interní** synchronizace před externí – může se tak dosáhnout vyšší úrovně paralelizace.
  Příklad – immutable fixed-sized hash table – zamykání se může aplikovat na jednotlivé buňky hash tabulky a není třeba zamykat celou tabulku -> vyšší míra paralelismu.

- Pokud metoda/třída spoléhá na **statickou proměnnou**, synchronizace se **se má řešit interně**.
  Příklad: `Math.random()` Kdyby nebyla synchronizace řešena interně, musela by se externě synchronizovat všechna volání `Math.random()` a stačilo by 1x zapomenout ...

Čekání na událost v multithreadovém provozu

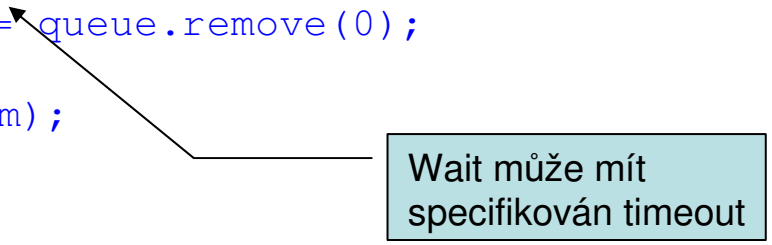
Perverzní konstrukce **busy-wait:**

```
private class WorkerThread extends Thread {
   public void run() {
      while (true) {
         synchronized (queue) {
            while (true) {
               if (!queue.isEmpty()) {
                  Object workItem = queue.remove(0);
                  break;
               }
            }
         processItem(workItem);
      }
   }
}
```

Procesor je při čekání vytížen, aniž by dělal cokoliv užitečného.

Čekání na událost v multithreadovém provozu

```
private class WorkerThread extends Thread {
   public void run() {
      synchronized (queue) {
         while (queue.isEmpty()) {
            queue.wait()
         }
         Object workItem = queue.remove(0);
      }
      processItem(workItem);
   }
}
```



WorkerThread čeká, dokud jiný thread nezavolá `queue.notify()` nebo `queue.notifyAll()`.

Při vstupu do `wait()` uvolní daný thread monitor (jinak by jiný thread nemohl zavolat `notify()`) a pak se tento thread zařadí se wait listu daného monitoru mezi ostatní čekající thready.

`notify()` způsobí, že některý ze threadů v daném wait listu se vzbudí.

`notifyAll()` vzbudí všechny theredy v daném wait listu.

Po vzbuzení musí vzbuzený thread získat zpět příslušný monitor. Znamená to tedy, že `notify(All)()` nemusí způsobit okamžité pokračování vzbuzeného threadu.

`wait(arg)` argument určuje timeout.

Čekání na událost v multithreadovém provozu

Vzhledem k tomu, že se při vstupu do `wait()` thread dočasně vzdá monitoru, mohou do synchronizovaného bloku vstupovat jiné thready. Před vstupem do `wait()` musí tedy být stav synchronizovaného objektu konzistentní.

Dogma:

Vždy používej `notifyAll()` místo `notify()`.

Opodstatnění:

Všechny čekající thready budou vzbuzeny a obslouženy. Nemůže se tedy stát, že bychom zapomněli probudit některý čekající thread.

Námitka:

Pokud čeká **n** threadů připravených k pokračování, ale pokračovat může jenom jeden (libovolný) z nich, pak jsem **n-1** theradů vzbudil zbytečně a ony mudou muset opět usínat. Složitost spojená s režií bude $O(n^2)$. V tomto případě `notifyAll()` není dobrá myšlenka, neboť notify() bude mít režii $O(n)$.
Příklad: semafor – čeká n threadů, libovolný, ale jenom jeden z nich, (jsou rovnocenné) může pokračovat.

Čekání na událost v multithreadovém provozu

Zásada:

`wait()` používej vždy v cyklu.

```
synchronized (queue) {
   while (queue.isEmpty()) {
      queue.wait()
   }
   ...
}
```

Opodstatnění:

Blokační podmínku je třeba testovat nejen

- při vstupu do `wait()`, ale i

- po opuštění `wait()`,
  neboť se mohla hodnota blokační podmínky změnit nezávisle na daném threadu:
  - Nezapomínejme, že uvnitř wait() daný thread nemá zamčený monitor, do kritické sekce tedy mohly vstoupit i další thready, které mohly hodnotu blokační podmínky změnit!
  - vzbuzeno mohlo být více threadů současně pomocí `notifyAll()`, jen jeden znich mohl změnit hodnotu blokační podmínky, ostaní vzbuzené thready ji musí znovu testovat a eventuálně znovu usnout.
  - v případě `wait()` s timeoutem mohlo k opuštění wait() dojít v důsledku vypršení timeoutu.

Zásady - doporučení

Další zásady:

- Nepoužívejme deprecated metody třídy `Thread`
  - `stop()`
  - `suspend()` / `resume()`

- Nepoužívejme deprecated metodu `Runtime.runFinalizersOnExit()`
  To je **thread-hostile** metoda (viz dále)

Viz *http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html*

Zásady - doporučení

Další doporučení:

- Nespoléhejme na strategii rozvrhování threadů – je implementačně závislá a mění se s různými implementacemi JVM:
  - `Thread.yield()`
  - `Thread.setPriority()`

  Poznámka: Doporučeníhodné je použití `Thread.yield()` při testování, aby se odhalily souběhy, které se při normálním rozvrhování threadů nemusí projevit.

- Vyhýbejme se použití *thread groups*.
  Joshua Bloch: *Effective Java – programming Language Guide* říká, že *thread groups* byl nepodařený experiment. Dokonce `ThreadGroup` API není úplně threadově bezpečné.

Zásady - doporučení

Dokumentuj threadovou bezpečnost navržené třídy.

JavaDoc může ignorovat (záleží na verzi) `synchronized`, protože to je implementační detail, nikoliv součást API.

Často používané úrovně threadové bezpečnosti:

- **immutable** – instance takové třídy se klientům jeví jako konstanty.
  Příklad: `String`, `Integer`, `BigInteger`
- **thread-safe** – instance takové třídy jsou mutable, ale všechny její (dostupné) metody mají dostatečnou interní synchronizaci
- **conditionally thread-safe** – threadově bezpečné s výjimkou některých metod, jež vyžadují externí synchronizaci.
  Příklad: `Hashtable`, `Vector`, jejichž iterátor vyžaduje externí synchronizaci
- **thread-compatible** – instance této třídy mohou být používány bezpečně za podmínky externí synchronizace každého vyvolání jejich metod.
  Příklad: ArrayList, HashMap mohou být externě synchronizovány pomocí `Collections.synchronizedList()` resp. `Collections.synchronizedMap()`
- **thread-hostile – např.** `System.runFinalizerOnExit()` - deprecated

Jiné způsoby zajištění threadové bezpečnosti

Od java 1.5 je dostupný package `java.util.concurrent`

Například:

```
java.util.concurrent.atomic.AtomicBoolean;
```

Má metody:

```
boolean
compareAndSet(boolean expect, boolean update)
```

Otestuje, zda je momentální hodnota dané instance rovna `expect` argumentu (tento výsledke nakonec vrátí jako návratovou hodnotu). Pokud ano, změní momentální hodnotu na hodnotu určenou `update` argumentem.
Test a nastavení se provede atomicky.

```
boolean getAndSet(boolean newValue)
```

Nastaví hodnotu instance na `newValue` a vrátí předchozí hodnotu.

Jiné způsoby zajištění threadové bezpečnosti

Příklady dalších tříd v `java.util.concurrent.atomic.*`

`AtomicBoolean`	
`AtomicInteger`	`int` hodnota, jež může být modifikována atomicky.
`AtomicIntegerArray`	`int` pole, jehož prvky mohou být modifikovány atomicky
`AtomicLong`	
`AtomicLongArray`	
`AtomicReference<V>`	reference na object, která může být modifikována atomicky.
`AtomicReferenceArray<E>`	pole referencí na objekty, jehož prvky mohou být modifikovány atomicky

Jiné způsoby zajištění threadové bezpečnosti

`java.util.concurrent.*`

Executors
a simple standardized interface for defining custom thread-like subsystems, including thread pools, asynchronous IO, and lightweight task frameworks.

Queues
ConcurrentLinkedQueue, LinkedBlockingQueue, ArrayBlockingQueue, SynchronousQueue, PriorityBlockingQueue, and DelayQueue.

Timing
The TimeUnit class provides multiple granularities (including nanoseconds) for specifying and controlling time-out based operations.

Synchronizers
Semaphore, CountDownLatch, Exchanger, ConcurrentHashMap, CopyOnWriteArrayList, and CopyOnWriteArraySet.

Concurrent Collections
ConcurrentHashMap, CopyOnWriteArrayList, and CopyOnWriteArraySet.