# Vícevláknové aplikace

## Obsah



# Základy

## Co jsou vlákna

**Vlákna** (mohli jsme také někde slyšet "lightweight procesy") jsou mechanismus, který umožňuje souběžné vykonávání více činností (výpočetních úkonů). Jedno vlákno reprezentuje právě jednu posloupnost operací, která je prováděna v rámci nějakého procesu. Každý *proces* (z pohledu operačního systému) musí mít minimálně jedno *vlákno*, ale může jich mít více. Tato vlákna pak mohou sdílet zdroje (proměnné, objekty, přístupy k zařízením) a je třeba zabývat se okolnostmi jejich současného běhu hlavně z pohledu přístupu k těmto společným zdrojům.

## Běh vláken

- Na jednoprocesorových, jednojádrových strojích běží současně právě jedno výpočetní vlákno. Iluze současného běhu je dosahováno rychlých přepínáním kontextů mezi vlákny, na němž vlákna nemusejí spolupracovat a o němž ani nemusejí vědět (preemptivní multitasking).
- Soudobé procesory s více jádry jdou dále, umožňují skutečně souběžnou práci více vláken.
- O spouštění vláken a přidělování kvant strojového času jim se stará plánovač procesů operačního systému. V Javě se do hry zapojuje samozřejmě hlavně JVM, která s plánovačem v OS spolupracuje.

# Vlákna v Javě

## Objekty a kód vláken

Vlákna jsou v Javě "uchopitelná" pomocí objektů třídy Thread.

Vytvoříme je tedy buďto:

- přímo rozšířením třídy Thread (vytvořením potomka)
- implementací rozhraní Runnable a předáním jeho instance při vytváření nového objektu Thread.

Každopádně jde o to napsat "někam" kód, který bude ve vlákně provádět.

## Příklad - přímé rozšíření 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();
// !!! nevolat přímo counterThread.run(); !!!
```

## Příklad - implementace Runnable a její použití

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

Runnable counter = new Counter();

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

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

# Synchronizace

## Proč a co?

- Synchronizovat přístup z více vláken musíme u všech společných (sdílených) zdrojů.
- Spočívá ve vyloučení souběžného přístupu ke sdílenému zdroji z více vláken a zajištění, že operace (metody) proběhnout atomicky (nepřerušeně vůči jiným vláknům nad týmž sdíleným zdrojem).
- Někde se o to implicitně postará třída sama - např. u starších kolekcí Java Core API (Vector, Stack, Hashtable, StringBuffer,...). Neplatí to ale pro všechny třídy Core API.

## Upozornění

Chyby v synchronizace jsou velice zákeřné a obtížně odhalitelné,
přitom v dnes obvyklých kontextech (např. víceuživatelské a webové aplikace) potenciálně hrozí vždy.

Obvykle se nedají reprodukovat ani odhalit testováním ani běžným laděním (např. pomocí debuggeru).
Lze je najít formálními postupy, které jsou však náročné.
Základem proto musí být PREVENCE a k té dospějeme jedině přes důkladné poznaní všech důležitých principů!

Nejlepší průpravou je v tomto případě dostatečné zvládnutí teorie.
Konkrétní výpočetní prostředí (Java, OS) pouze dotvářejí technický rámec - principy zůstávají.

## Kritické sekce a monitory

- Kritická sekce - je úsek výpočtu, v němž musí být zajištěn výlučný (nesouběžný) přístup ke sdílenému zdroji. Po dobu pobytu vlákna v kritické sekci nesmí stejnou kritickou sekci provádět žádné jiné vlákno.
- Monitor - je základním nástrojem pro synchronizaci. Má ho každý objekt a slouží k zamykání přístupu do kritické sekce.

## Synchronizace označením kritické sekce

- Pomocí klíčového slova `synchronized` lze označit celou metodu jako kritickou sekci. K zamčení se použije monitor objektu/instance, na níž se metoda volá. Jiná vlákna provádějící jiné `synchronized` metody nebo úseky kódu téhož objektu čekají na dokončení.
- Pomocí bloku `synchronized(o) { příkazy }` lze vyznačit jen dílčí úsek metody a k synchronizaci použít monitor objektu `o`. Vlákna synchronizovaná týmž zámkem čekají.

## "synchronized" metoda

```
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;
    }
}
```

### Poznámka

Celá metoda se zamkne monitorem objektu "this" - vždy.

# "synchronized" blok

```
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;
        }
    }
}
```

### Poznámka

Blok se zamkne se monitorem objektu "this" - mohli bychom i jiným.

# Kdy je synchronizace zbytečná

Nadbytečná synchronizace může zejména na moderních architekturách představovat zbytečnou zátěž navíc a zpomalení běhu aplikace. Proto bychom měli synchronizovat právě to, co je třeba. *Co není třeba synchronizovat?*

- Při přístupu k nesdíleným proměnným a objektům (např. neviditelným z jiných vláken)
- Při volání metod tříd, které jsou již synchronizované (vláknově bezpečná, thread-safe)
- Při přístupu ke `static final` objektům (konstantám) od verze Java 5 výše
- Při přístupu k lokálním proměnným - i proto bychom je měli používat všude tam, kde atributy nejsou nezbytně nutné! Lokální proměnné jsou uloženy na zásobníku vlákna. Ten patří právě jednomu vláknu a nejsou tedy sdíleny mezi více vlákny. Lokální proměnná nicméně může obsahovat referenci na objekt na haldě (prostoru dynamické paměti), který už může být sdílený. Nemusíme tedy synchronizovat *manipulaci* s hodnotou lokální proměnné, ale pro manipulaci s odkazovaným *objektem* už to platit nemusí!!!

# Časté chyby v sychonizaci

## Příklad 1 (viz Wiki)

```
class Counter {
    private static int currentValue = 0;
    public synchronized int next() {
```

```
        return ++currentValue;
    }
}
```

# Chyba - synchronizovat "málo"

```
class Counter {
    private static int currentValue = 0;
    public synchronized int next() {
        return ++currentValue;
    }
}
```

### Poznámka

Při volání `next` se zamyká pouze instance `Counter`, nikoli sdílená hodnota (protože ta je `static`, společná všem objektům `Counter`).

# Příklad 2 (viz Wiki)

```
class Counter {
    private int currentValue = 0;
    private static Object LOCK = new Object();
    public int next() {
        synchronized(LOCK) {
            return ++currentValue;
        }
    }
}
```

# Chyba - synchronizovat "moc"

```
class Counter {
    private int currentValue = 0;
    private static Object LOCK = new Object();
    public int next() {
        synchronized(LOCK) {
            return ++currentValue;
        }
    }
}
```

### Poznámka

Při volání `next` se nesprávně zamyká přes LOCK, který je společný všem instancím `Counter`, ostatní "Countery" budou zbytečně blokovány.

# Příklad 3 (viz Wiki)

```
public class SomeThreadSafeClass {

    private List<String> data = Collections.synchronizedList(new ArrayList<String>
```

```
    public void addData(String newObject) {
        data.add(newObject);
    }

    public synchronized void dumpData() {
        for(String s : data) {
            System.out.println(s);
        }
    }

}
```

## Chyba - synchronizovat "neúplně"

```
public class SomeThreadSafeClass {

    private List<String> data = Collections.synchronizedList(new ArrayList<String>

    public void addData(String newObject) {
        data.add(newObject);
    }

    public synchronized void dumpData() {
        for(String s : data) {
            System.out.println(s);
        }
    }

}
```

### Poznámka

Chybou je synchronizovat jen `dumpData` a ne `addData`, protože takto během iterace v `dumpData` klidně proběhne (nevyloučené) vložení do kolekce `data` a ta se při iteraci stane nekonzistentní.

# Další zdroje

## Sun

Sun Java tutoriál, kapitoly k souběžnému počítání (concurrency)

| | |
|---|---|
| Concurrency in Swing | http://java.sun.com/docs/books/tutorial/uiswing/concurrency/index.html |
| Threads and Swing Components | http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html |

## FI

Wiki FI:

- Téma Vlákna [http://kore.fi.muni.cz:5080/wiki/index.php/PV168/Vl%C3%A1kna]