



Vícevláknový přístup



Použití vláken

- MS Word: píšete text a na pozadí se provádí kontrola pravopisu.
 - Kontrola pravopisu běží uvnitř vlákna.
- Total Commander: když kopírujete soubor z jednoho adresáře do druhého, pak TC „nezamrzne“, zobrazuje průběh kopírování a kopírování souborů můžete i přerušit.
 - Kopírování souborů běží uvnitř vlákna.
- Přehrávač hudby: Aplikace přehrává hudbu a vy tuto operaci můžete pozastavit a celkově s aplikací pracovat, zatímco aplikace přehrává hudbu dál.
 - Přehrávání hudby běží uvnitř vlákna.



Vlákno

- Vlákno v Javě je třída, která:
 - Buď je potomkem třídy Thread.
 - Nebo implementuje rozhraní Runnable.
- Na následujících dvou snímcích oba způsoby prezentuji a budu spíš používat první způsob protože je kratší na způsob zápisu, druhý způsob je ale flexibilnější.
- V obou případech vlákno implementuje metodu `run()`, která se zavolá po spuštění vlákna.
- Jakmile metoda `run()` doběhne (nebo se z ní vyhodí výjimka), pak se vlákno ukončí.
- Vlákno se spouští voláním metody `thread.start()`



Vlákno pomocí třídy Thread

- Vlákno:

```
public class HelloThread extends Thread {  
    public void run() {  
        System.out.println("hello from thread");  
    }  
}
```

- Spuštění vlákna:

```
HelloThread thread = new HelloThread();  
System.out.println("start thread");  
thread.start();  
System.out.println("thread started");
```

- Výsledek:

```
start thread  
thread started  
hello from thread
```



Vlákno pomocí interface Runnable

- Vlákno:

```
public class HelloThread implements Runnable {  
    public void run() {  
        System.out.println("hello from thread");  
    }  
}
```

- Spuštění vlákna:

```
System.out.println("start thread");  
Thread thread = new Thread(new HelloThread());  
thread.start();  
System.out.println("thread started");
```

- Výsledek je stejný.

Java 8:

```
Thread thread = new Thread(() -> {  
    System.out.println("hello from thread");  
}).start();
```



Zajímavé atributy třídy Thread

- `Thread.sleep(1000)`: statická metoda, která uspí aktuální vlákno na definovaný počet milisekund.
 - 1000 milisekund = 1 sekunda.
- `setName()`: jméno vlákna. Toto jméno je vidět například v nástroji jvisualvm.
- `setPriority()`: priorita vlákna. Může nabývat hodnot:
 - `Thread.NORM_PRIORITY`: Výchozí hodnota.
 - `Thread.MAX_PRIORITY`: Vlákno dostane přiděleno maximum strojového času (příklad: vlákno na kopírování souborů v Total Commanderu).
 - `Thread.MIN_PRIORITY`: Vlákno dostane přiděleno minimum strojového času (příklad: kontrola pravopisu v MS Word).



Daemon

- Daemon je typ vlákna, které slouží pro podpůrné činnosti v aplikaci (například kontrola pravopisu ve Wordu).
- Nově vytvořené vlákno automaticky není daemon. Stane se tím po zavolání `thread.setDaemon(true)`:

```
HelloThread thread = new HelloThread();  
thread.setDaemon(true);
```

- Běh celé aplikace se ukončí, když neběží žádné vlákno, které není daemon.



interrupt, join

- Když je vlákno uspané pomocí metody `sleep()`, jiné vlákno na něm může zavolat `thread.interrupt()`, což vyvolá výjimku typu `InterruptedException`. Je best practice, aby vlákno v tom případě ukončilo svoji činnost.
- Je možné zavolat metodu `thread.join()`, která pozastaví vykonávání aktuálního vlákna, dokud vlákno `thread` neukončí svoji činnost.



Příklad

```
public class HelloThread extends Thread {  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(getName());  
            try { sleep(100); } catch (InterruptedException ex) { }  
        }  
    }  
}
```

```
HelloThread jamaica = new HelloThread();  
jamaica.setName("Jamaica");  
jamaica.start();  
HelloThread fiji = new HelloThread();  
fiji.setName("Fiji");  
fiji.start();
```



Výsledky

Jednotlivé běhy vláken. Není možné přesně určit, jaké vlákno se v jakou dobu zavolá. Pro zobrazení stavu vlákna je vhodné použít aplikaci jvisualvm.

Jamaica	Jamaica	Jamaica	Jamaica
Fiji	Fiji	Fiji	Fiji
Jamaica	Jamaica	Jamaica	Jamaica
Fiji	Fiji	Fiji	Fiji
Fiji	Fiji	Fiji	Jamaica
Jamaica	Jamaica	Jamaica	Fiji
Fiji	Fiji	Fiji	Fiji
Jamaica	Jamaica	Jamaica	Jamaica
Fiji	Fiji	Fiji	Jamaica
Jamaica	Jamaica	Jamaica	Fiji
Jamaica	Jamaica	Fiji	Fiji
Fiji	Jamaica	Jamaica	Jamaica
Fiji	Fiji	Fiji	Jamaica
Jamaica	Jamaica	Jamaica	Fiji
Fiji	Jamaica	Fiji	Jamaica
Jamaica	Fiji	Jamaica	Fiji
Fiji	Fiji	Fiji	Jamaica
Jamaica	Jamaica	Jamaica	Fiji

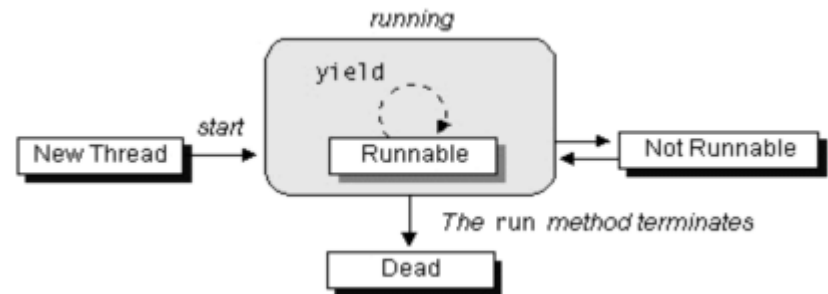
Životní cyklus vlákna

Vlákno se stává
Not Runnable
v následujících
případech:

Zavolá se metoda
`sleep()`

Vlákno zavolá metodu
`wait()`, aby čekalo na
specifickou
podmínku

Vlákno je zablokované
na I/O





Synchronizace I.

- Mějme tuto třídu:

```
public class Counter {  
    private int count = 0;  
  
    public void inc() {  
        count++;  
    }  
  
    public void dec() {  
        count--;  
    }  
  
    public int getCount() {  
        return count;  
    }  
}
```

- Operace `count++` (metoda `inc()`) se skládá ze tří kroků:
 - Získej aktuální hodnotu `count`
 - Incrementuj hodnotu o 1.
 - Ulož výsledek do atributu `count`
- Operace `count--` funguje obdobně.
- Když jedno vlákno zavolá metodu `inc()` a druhé současně zavolá metodu `dec()`, pak může dojít k následujícímu chování:
 - Vlákno 1: Získej `count`.
 - Vlákno 2: Získej `count`.
 - Vlákno 1: Inc. `count`, výsledek: 1
 - Vlákno 2: Dec. `count`, výsledek: -1
 - Vlákno 1: Ulož výsledek, `count` = 1
 - Vlákno 2: Ulož výsledek, `count` = -1



Synchronizace II.

- Chyba jako na předcházejícím snímku se špatně hledá a opravuje (protože se vyskytuje pouze když dvě vlákna operují se stejnými atributy).
- Řešením je synchronizace, což je prakticky způsob jak z paralelního běhu udělat sériový. Na druhou stranu, kdybychom synchronizaci používali všude, pak by vlákna neměla význam.
- Existují dva způsoby jak synchronizovat:
 - Synchronizace na úrovni metod.
 - Synchronizace uvnitř bloku.



Synchronizace – příklad I.

```
public class IncThread extends Thread {
```

```
    private Counter counter;
```

```
    public IncThread(Counter counter) {  
        this.counter = counter;  
    }
```

```
    public void run() {  
        for (int i = 0; i < 100000; i++) {  
            counter.inc();  
        }  
    }
```

```
}
```



Tisíckrát přičteme jedničku



Synchronizace – příklad II.

```
public class DecThread extends Thread {
```

```
    private Counter counter;
```

```
    public DecThread(Counter counter) {  
        this.counter = counter;  
    }
```

```
    public void run() {  
        for (int i = 0; i < 100000; i++) {  
            counter.dec();  
        }  
    }
```

```
}
```



Tisíckrát odečteme jedničku

Synchronizace – příklad III.

```
public class Main {
```

```
    public static void main(String[] args) throws Exception {
```

```
        Counter counter = new Counter();
```

```
        IncThread incThread = new IncThread(counter);
```

```
        incThread.start();
```

```
        DecThread decThread = new DecThread(counter);
```

```
        decThread.start();
```

```
        incThread.join();
```

```
        decThread.join();
```

```
        System.out.println(counter.getCount());
```

```
    }
```

```
}
```

Počkáme na dokončení obou vláken.

Pokud by byla synchronizace provedená dobře, pak by měl být výsledek nula.

Poznámka: Podobný příklad trochu úsporněji zapsaný pomocí lambda výrazů (Java 8) je zde:

<https://gist.github.com/jirkapinkas/d9907acf796de235193f61256e5eed12>



Synchronizace – řešení I.

- Kdybychom použili synchronizaci na úrovni metody, pak bychom museli na každé metodě, kde pracujeme s atributem count, nastavit klíčové slovo synchronized:

```
public class Counter {  
    private int count = 0;  
  
    public synchronized void inc() {  
        count++;  
    }  
  
    public synchronized void dec() {  
        count--;  
    }  
  
    public synchronized int getCount() {  
        return count;  
    }  
}
```



Synchronizace – řešení II.

- Nebo klíčové slovo `synchronized` nepoužijeme na metodách cílové třídy, ale tam, kde tyto metody vyvoláváme:
- V metodě `run()` ve třídě `IncThread`:

```
synchronized (counter) {  
    counter.inc();  
}
```
- V metodě `run()` ve třídě `DecThread`:

```
synchronized (counter) {  
    counter.dec();  
}
```



Synchronizace – zámky I.

- Synchronizace ke své činnosti používá zámek objektu (v anglické terminologii lock, nebo monitor). Každý objekt má jeden zámek. Každé vlákno, které chce získat k nějakému objektu exkluzivní přístup (čehož se zajišťuje pomocí synchronizace), potřebuje získat tento zámek, aby s daným objektem mohlo pracovat.



Synchronizace – zámky II.

- V případě synchronizace pomocí bloku je jasné, na jakém objektu je zámek získáván:

```
synchronized (counter) {  
    counter.inc();  
}
```

↖ Jedná se o tento objekt

- V případě synchronizace celé metody to nemusí být zřejmé. Tyto dvě metody jsou stejné:

```
public synchronized void inc() { count++; }
```

```
public void inc() {  
    synchronized (this) { count++; }  
}
```



Synchronizace – zámky III.

- Poznámka: V případě synchronizace statické metody se synchronizuje celá třída. Tyto dvě metody jsou stejné:

```
public synchronized static void inc() {  
}
```

```
public static void inc() {  
    synchronized (Counter.class) {  
  
    }  
}
```

Poznámka: Toto je nejjednodušší případ synchronizace, mnohem lepší řešení budou následovat.



Synchronizace – Lombok

- Alternativně je možné s Lombokem použít anotaci `@Synchronized`, která dělá víceméně to samé, ale pro synchronizaci se používá další objekt, jehož jedinou rolí je řešit synchronizaci. Princip, který se používá je hodně podobný tomu, co dělá `Collections.synchronizedCollection()`:
 - <https://projectlombok.org/features/Synchronized.html>
 - <http://greppcode.com/file/repository.greppcode.com/java/root/jdk/openjdk/6-b14/java/util/Collections.java#Collections.SynchronizedCollection.0mutex>



ReentrantLock

- Alternativou k synchronized bloku je ReentrantLock:
 - <https://stackoverflow.com/questions/11821801/why-use-a-reentrantlock-if-one-can-use-synchronizedthis>
- Pomocí lock.lock() získá zámek, pomocí lock.unlock() uvolní zámek. Navíc ale obsahuje řadu užitečných metod.
- Nebo ReentrantReadWriteLock, což je lock, který umožňuje více vláknům číst, ale jenom jedno vlákno může zapisovat:
 - <http://tutorials.jenkov.com/java-util-concurrent/readwritelock.html>



Thread Starvation

- K vyhladovění vlákna dojde když nějaké vlákno nemá šanci se zavolat. Příklad:
 - <https://gist.github.com/jirkapinkas/7985f079d6f91e4db4a1c7303007d6fe>
- Řešení: Fair Reentrant lock
 - <http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html>



Synchronizace – nejlepší řešení I.

- Úplně nejlepší řešení je, abychom synchronizaci neřešili my sami, ale aby ji řešil někdo jiný. V Javě existuje několik užitečných tříd, které mají atomické operace anebo jsou jejich operace synchronizované:
- `AtomicInteger`, `AtomicBoolean`, `AtomicLong`
- `Collections.synchronizedList()`,
`Collections.synchronizedSet()`,
`Collections.synchronizedCollection()`
- `ConcurrentHashMap`
 - Proč používat `ConcurrentHashMap` a nepoužívat `Collections.synchronizedMap()`:
<https://stackoverflow.com/questions/510632/whats-the-difference-between-concurrenthashmap-and-collections-synchronizedmap>



Synchronizace – nejlepší řešení II.

- Ve výjimečných situacích je možné použít `CopyOnWriteArrayList` a `CopyOnWriteArraySet`.
- Pro implementaci fronty se používá interface `BlockingQueue`:
 - <http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html>



Thread-safety

- Psaní thread-safe kódu je zejména o řízení přístupu ke stavu objektu (a zejména o řízení přístupu ke stavu, který se po dobu života objektu mění).
- Stateless objekty jsou vždy thread-safe.
- Když má objekt atribut, kterou vlákna mění (mutable state variable), pak existují tři možnosti, jak to vyřešit:
 - Nesdílet atribut mezi vlákny.
 - Z mutable atributu vytvořit immutable atribut.
 - Při přístupu k atributu použít synchronizaci.
- Pro implementaci výše uvedeného je důležité dodržovat principy dobrého OOP – encapsulation (zapouzdření) a immutability.



Immutable & Thread-safe

- Immutable objekt (objekt, jehož obsah je po jeho konstrukci neměnný – například String) je thread-safe. Pozor ale na to, aby byl objekt skutečně immutable!!!
 - <https://dzone.com/articles/do-immutability-really-means>
- Nejlepší je hledat jestli je nějaká třída thread-safe. Například:
 - „is JdbcTemplate thread safe“



Java Concurrency In Practice

- JCIP (Java Concurrency In Practice) je nadčasová knížka, ze které bude následovat několik příkladů.
- Poznámka: V knížce se používají anotace jako je `@NotThreadSafe` apod. Nejedná se o standardní anotace, ale je možné je použít k lepší dokumentaci tříd.
- Pro jejich použití přidejte tuto dependency:
 - <https://javalibs.com/artifact/com.google.code.findbugs/jsr305>

Singleton Lazy initialization



```
@NotThreadSafe
public class LazyInitRace {
    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
        if (instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}
```

- Pro bezpečné vytvoření Singletonu existuje několik technik (nicméně osobně preferuji Spring):
- <https://stackoverflow.com/questions/11165852/java-singleton-and-synchronization>

Reordering



```
public class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready)
                Thread.yield();
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

- Výsledkem běhu tohoto kódu může být:
 - 42
 - Zacyklení
 - 0 ???
- Není garantováno, že se operace v jednom vlákně provedou ve stejném pořadí, jako jsou v kódu (tento fenomén má název Reordering).

Immutable



```
class UnsafeStates {  
    private String[] states = new String[] {  
        "AK", "AL" ...  
    };  
    public String[] getStates() { return states; }  
}
```

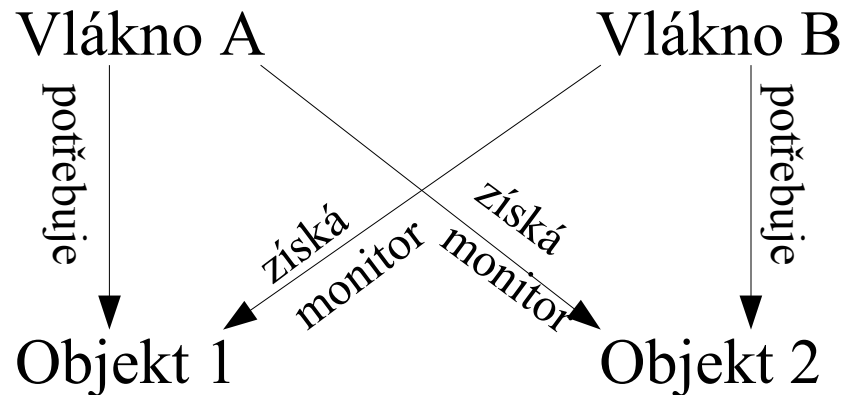
```
@Immutable  
public final class ThreeStooges {  
    private final Set<String> stooges = new HashSet<String>();  
  
    public ThreeStooges() {  
        stooges.add("Moe");  
        stooges.add("Larry");  
        stooges.add("Curly");  
    }  
  
    public boolean isStooge(String name) {  
        return stooges.contains(name);  
    }  
}
```

- Vytvořit immutable objekt nemusí být úplně jednoduché. Například první příklad není immutable, protože je možné zvnějšku změnit jeho vnitřní stav.

<http://jcip.net/listings/UnsafeStates.java>
<http://jcip.net/listings/ThreeStooges.java>

Deadlock

- Pokud se dostanete do této situace, pak jste se dostali do deadlocku:



- Poznámky:
 - Řešením je přeprogramování příslušných metod.
 - Deadlock zjistíte pomocí jvisualvm.



Deadlock - příklad

- Příklad:

- <https://gist.github.com/jirkapinkas/ce1285df9a06d7ba094eb78997942563>



Klíčové slovo volatile

- Příklad: Jedno vlákno mění hodnotu atributu counter. Druhé vlákno tuto hodnotu průběžně čte. Ve výchozím nastavení vlákno může zkopírovat hodnotu atributu z operační paměti do CPU cache kde na ní poté provádí operace.
- Průšvih je v tom, že druhé vlákno čte hodnotu z operační paměti (která tudíž nemusí odpovídat skutečnosti).
- Řešení je u atributu přidat klíčové slovo volatile, které garantuje, že po změně atributu v CPU cache se tato změna ihned promítne do operační paměti:
 - <http://tutorials.jenkov.com/java-concurrency/volatile.html>



wait(), notify(), notifyAll()

- Běh vláken je možné navzájem koordinovat pomocí metod:
 - `wait()`: uspí aktuální vlákno (bude čekat do probuzení zavoláním metody `notify()` nebo `notifyAll()`).
 - `notify()`: probudí jedno vlákno, které čeká na probuzení na nějakém objektu.
 - `notifyAll()`: probudí všechna vlákna, která čekají na probuzení na nějakém objektu.
- K čemu se to hodí? Pro Producer / Consumer vlákna. Consumer čeká. Producer uloží data do sdíleného objektu a probudí Consumera. Ten tyto data odebere, zpracuje a opět čeká. A takto neustále dokolečka.



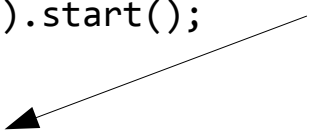
Executor

- Vlákna používaná dosud prezentovaným způsobem jsou super, pokud potřebujete pracovat pouze s jednotkami vláken a nepotřebujete scheduling (periodické vykonávání metody v časovém intervalu). Pro tyto pokročilejší vlastnosti se od Java SE 5 používá Executor.
- Tyto operace jsou stejné:

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

```
Executor executor = Executors.newSingleThreadExecutor();  
executor.submit(new HelloThread());
```

V této třídě je řada metod pro vytvoření různých typů executorů.





Future

- `ExecutorService.submit()` akceptuje dva typy operací:
 - `Runnable` s metodou `void run()`
 - `Callable` s metodou `V call()` (`Callable` vrátí nějaký objekt)
- Dále metoda `submit()` vrátí objekt typu `Future`, který zejména obsahuje metodu `get()`, která počká až doběhne běh operace a vrátí výsledek (pro `Runnable` vrátí `null`).



Scheduled Executor

- Tímto způsobem se bude metoda `run()` vlákna `HelloThread` volat periodicky každé dvě vteřiny:

```
ScheduledExecutorService scheduledExecutor =  
    Executors.newSingleThreadScheduledExecutor();  
scheduledExecutor.scheduleAtFixedRate  
    (new HelloThread(), 0, 2, TimeUnit.SECONDS);
```

- K čemu je to dobré? Já například všem účastníkům školení dva týdny před zahájením školení tímto způsobem rozesílám emaily s připomínkou brzkého konání školení.
- Je to vlastně cron uvnitř Java procesu :-)



Fork / Join

- Od Java SE 7 je standardní součástí Java SE fork / join knihovna.
- Prakticky pokud máte časově náročnou úlohu a můžete ji rozdělit do menších částí, které je možné vyřešit odděleně a máte procesor s více jádry, pak je toto velice efektivní způsob, jak je využít.
 - <http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html>
 - Příklad:
 - <https://gist.github.com/jirkapinkas/9f5d75ddf734b0910fc7c5ae3ec52a58>



Semaphore

- Semaphore je způsob, jak zabezpečit:
 - aby do nějaké metody v jednu chvíli mohl vstoupit omezený počet vláken,
 - nebo pro posílání signálů mezi dvěma vlákny (semaphore o velikosti 1).
- <http://tutorials.jenkov.com/java-util-concurrent/semaphore.html>
- Poznámka: Semaphore má konstruktor, který umožňuje nastavit fairness.



CountDownLatch

- CountDownLatch umožňuje jednomu nebo více vláknům čekat, až doběhne sada operací. Při vytvoření se CountDownLatch inicializuje s výchozím počtem, který se metodou `countDown()` snižuje a jakmile se sníží na nulu, tak se probudí vlákno, které zavolalo `await()` pro čekání na tuto událost.
 - <http://tutorials.jenkov.com/java-util-concurrent/countdownlatch.html>



Cyclic Barrier

- Cyklická bariéra je synchronizační mechanismus, ve kterém musí stanovený počet vláken dosáhnout nějakého bodu, aby mohly všechna vlákna pokračovat dál.
 - Pozor na to, že to je velice náchylné k logickým chybám. Osobně před cyklickou bariérou preferuji uložit objekty typu Future do listu a poté ve foreach cyklu na nich zavolat future.get().
 - <http://tutorials.jenkov.com/java-util-concurrent/cyclicbarrier.html>



Exchanger

- Exchanger je synchronizační mechanismus, kdy si dvě vlákna mezi sebou vyměňují objekty:
 - <http://tutorials.jenkov.com/java-util-concurrent/exchanger.html>