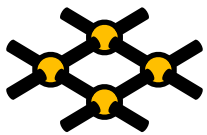
Vláknové programování
část V

Lukáš Hejmánek, Petr Holub
{xhejtman,hopet}@ics.muni.cz



Laboratoř pokročilých síťových technologií

PV192
2010–03–23

Přehled přednášky

Pokročilé vlasnosti Javy
- Atomické typy
- Concurrent Collections
- Explicitní zamykání
- Executors, Thread Pools a Futures

Atomické typy

- čekání na **`synchronized`** monitor vede na přeplánování vlákna
- atomické proměnné to zvládnou bez přepínání kontextu
  - vyšší výkon pro nízkou až střední míru soutěžení o zámek (lock contention)
  - tzv. wait-free synchronizace

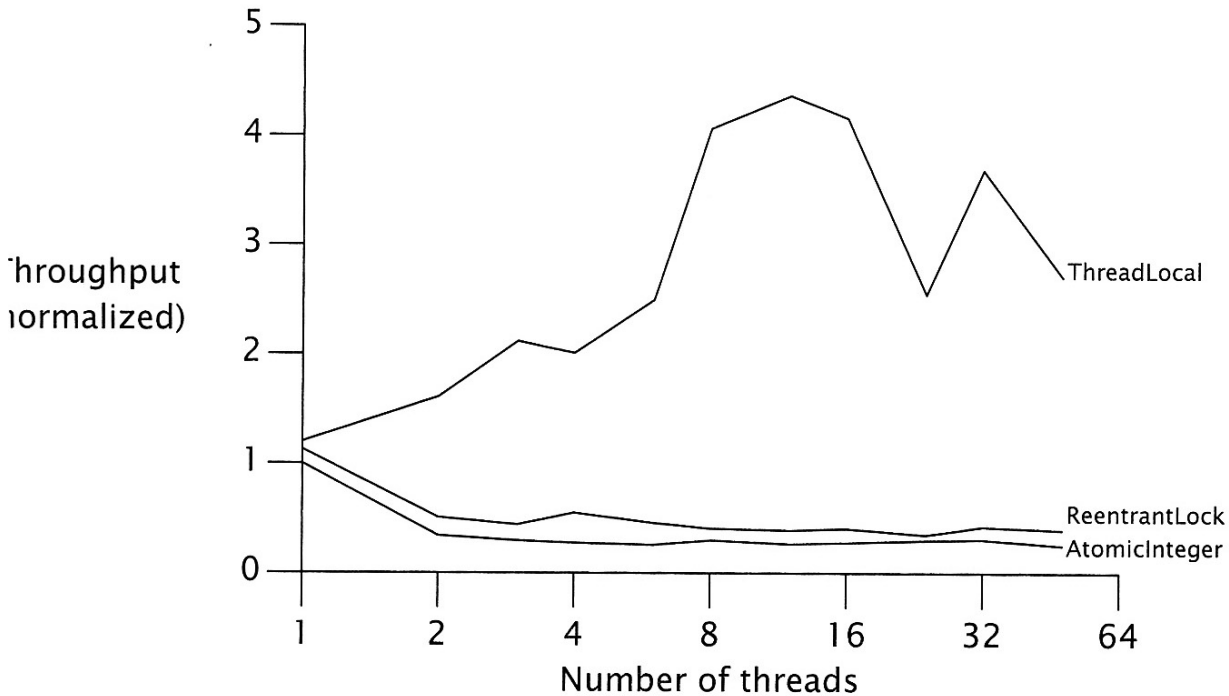


FIGURE 15.1. Lock and `AtomicInteger` performance under high contention.

Zdroj: Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D. *Java Concurrency in Practice.* Addison Wesley Professional, 2006

Atomické typy

- čekání na **`synchronized`** monitor vede na přeplánování vlákna
- atomické proměnné to zvládnou bez přepínání kontextu
  - vyšší výkon pro nízkou až střední míru soutěžení o zámek (lock contention)
  - tzv. wait-free synchronizace

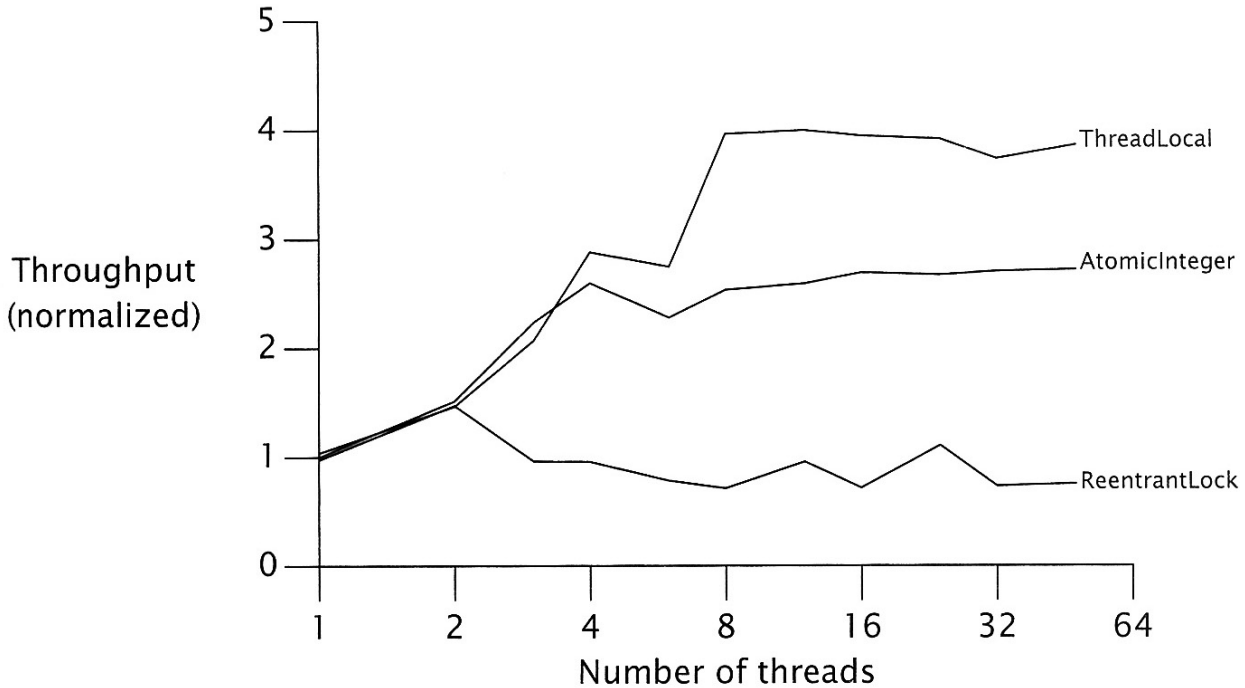


FIGURE 15.2. Lock and `AtomicInteger` performance under moderate contention.

Zdroj: Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D. *Java Concurrency in Practice.* Addison Wesley Professional, 2006

Atomické typy

- podpora v HW
  - *compare-and-swap* (CAS)
    - CAS(x, y)
    - funkce: porovnej obsah paměti s x a pokud je identický, nahraď jej za y
    - návratová hodnota: úspěch změny (buď jako **`boolean`** nebo jako hodnota, kterou má paměť *před* provedením instrukce)
    - podpora: IA-32, Sparc
  - *double compare-and-swap* (DCAS/CAS2)
    - funkce: výměna hodnot na dvou místech v paměti na základě původních hodnot
    - jednoduchá implementace atomické Deque
    - lze emulovat pomocí CAS $\Longrightarrow$ (pomalá) podpora u Motorol 68k
  - *double-wide compare-and-swap*
    - funkce: výměna hodnot na dvou přilehlých místech v paměti
    - podpora: **`CMPXCHG8B`** a **`CMPXCHG16B`** na novějších x86
  - *Single compare, double swap*
    - funkce: výměna hodnot na dvou místech v paměti v závislosti na jedné původní hodnotě
    - podpora: **`cmp8xchg16`** u Ithania

Atomické typy

- podpora v HW
  - *load-link/store-conditional* (LL/SC)
    - funkce: (1) LL načte hodnotu paměti, (2) SC ji změní pouze pokud se původní hodnota od operace LL nezměnila, jinak selže
    - silnější než CAS – řeší i problém ABA
    - podpora: **`ldl_l/stl_c`** a **`ldq_l/stq_c`** (Alpha), **`lwarx/stwcx`** (PowerPC), **`ll/sc`** (MIPS), **`ldrex/strex`** (ARM version 6 avyšší)
  - *fetch-and-add*
    - funkce: atomická inkrementace obsahu paměti
    - návratová hodnota: původní hodnota paměti
    - podpora: x86 od 8086 (**`ADD`** s prvním operandem specifikujícím místo v paměti, nicméně nevrací původní hodnotu – s LOCK prefixem atomické i u více procesorů), XADD od 486 vrací původní hodnotu

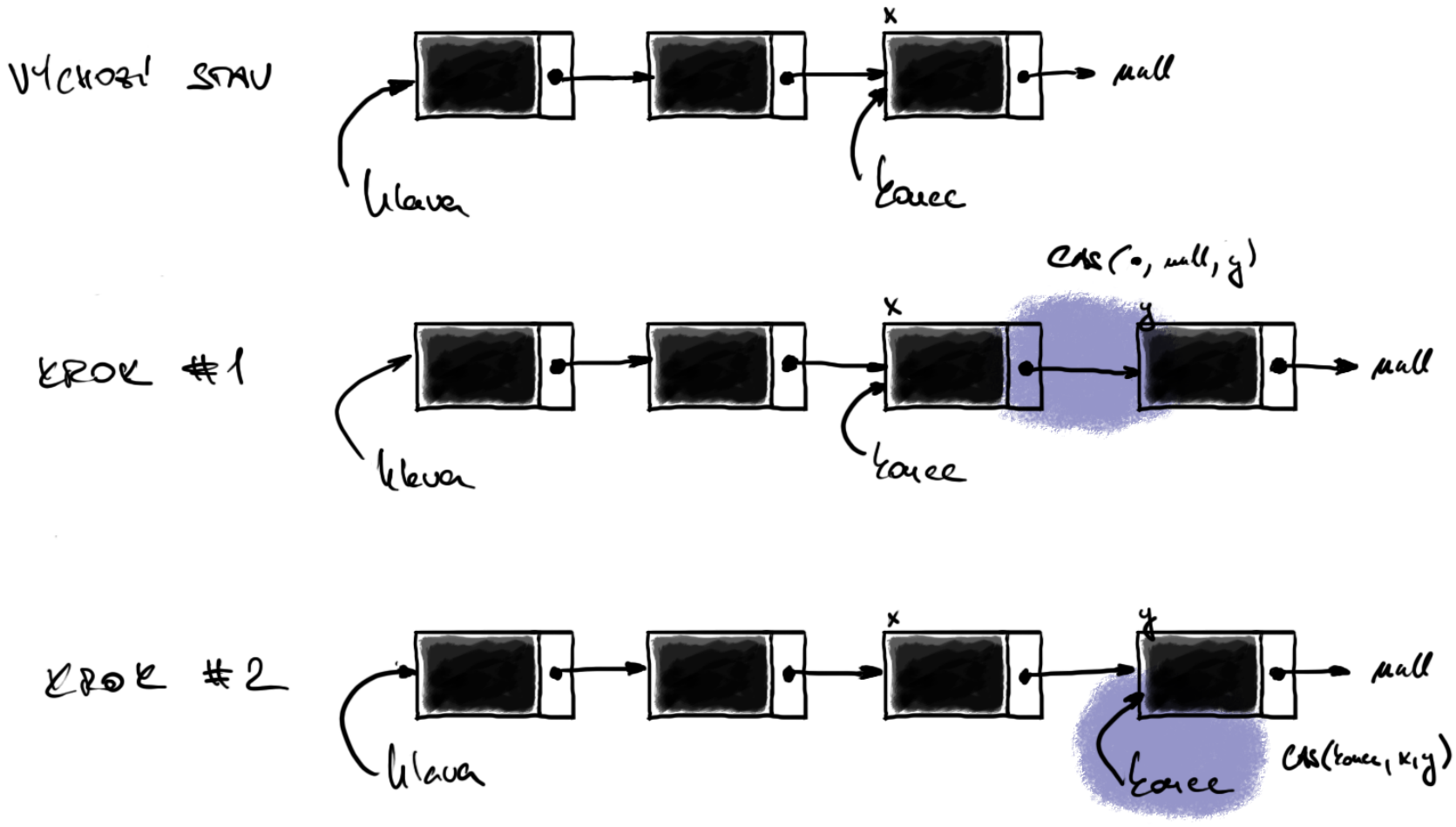
Atomické typy

- **`AtomicX`** z **`java.util.concurrent`**
  - **`AtomicBoolean`**, **`AtomicInteger`**, **`AtomicLong`**, **`AtomicReference`**
- Zajištěné atomické aktualizace
- Podpora od Java 5.0
- HW optimalizace
  - CAS instrukce (IA-32, Sparc)
  - podpora v JVM od 5.0

Využití atomických typů

- Návrh algoritmu
  - buď vyžaduje pouze jednu atomickou změnu
  - nebo z první změny musí být odvoditelné ostatní a musí je být schopen dokončit „kdokoli“
- Kolekce
  - ConcurrentLinkedQueue
  - WaitFreeReadQueue
    `http://www.rtsj.org/specjavadoc/javax/realtime/WaitFreeReadQueue.html`
  - WaitFreeWriteQueue
    `http://www.rtsj.org/specjavadoc/javax/realtime/WaitFreeWriteQueue.html`

Neblokující seznam: Michael-Scott, 1996



Neblokující seznam: Michael-Scott, 1996

```
1  import java.util.concurrent.atomic.AtomicReference;
   // dle http://www.javaconcurrencyinpractice.com/listings/LinkedQueue.java
3
   public class AtomickySeznam<E> {
5      private static class Node<E> {
           final E polozka;
7          final AtomicReference<AtomickySeznam.Node<E>> next;

9          public Node(E polozka, AtomickySeznam.Node<E> next) {
               this.polozka = polozka;
11             this.next = new
                       AtomicReference<AtomickySeznam.Node<E>>(next);
13         }
       }
15
       private final AtomickySeznam.Node<E> dummy =
17             new AtomickySeznam.Node<E>(null, null);
       private final AtomicReference<AtomickySeznam.Node<E>> hlava
19             = new AtomicReference<AtomickySeznam.Node<E>>(dummy);
       private final AtomicReference<Node<E>> konec
21             = new AtomicReference<AtomickySeznam.Node<E>>(dummy);
```

Neblokující seznam: Michael-Scott, 1996

```
22
    public boolean put(E polozka) {
24      AtomickySeznam.Node<E> newNode =
                new AtomickySeznam.Node<E>(polozka, null);
26      while (true) {
            AtomickySeznam.Node<E> curkonec = konec.get();
28          AtomickySeznam.Node<E> konecNext = curkonec.next.get();
            if (curkonec == konec.get()) {
30              if (konecNext != null) {
                    // dokoncime rozpracovany stav - posuneme konec
32                  konec.compareAndSet(curkonec, konecNext);
                } else {
34                  // pokusime se vlozit
                    if (curkonec.next.compareAndSet(null, newNode)) {
36                      // pri uspechu se pokusime posunout konec
                        konec.compareAndSet(curkonec, newNode);
38                      return true;
                    }
40              }
            }
42      }
    }
44
}
```

Problém ABA

- Problém, jak detekovat změnu $A \rightarrow B \rightarrow A$
  - podpora HW: LL/SC
  - „verzování“: počítadlo změn
- AtomicStampedReference
  - odkaz + **`int`** počítadlo změn
- AtomicMarkedReference
  - odkaz + **`boolean`** indikátor
  - některé algoritmy používají indikátor k označení uzlu v seznamu jako smazaného

Concurrent Collections

- optimalizace kolekcí na výkon při paralelních přístupech
- CopyOnWriteArrayList, CopyOnWriteArraySet
  - optimalizované pro režim čti-často-měň-zřídka
  - CopyOnWriteArraySet obdoba HashSet
  - CopyOnWriteArrayList obdoba ArrayList, na rozdíl od Vector poskytuje složené operace
  - iterace poskytuje pohled na objekt v době konstrukce iterátoru

```
1  import java.util.concurrent.*;

3  public class CoW {
       CopyOnWriteArraySet cowAS = new CopyOnWriteArraySet();
5      CopyOnWriteArrayList cowAL = new CopyOnWriteArrayList();
       public void narabaj() {
7          cowAS.addAll(kolekce);
           cowAS.contains(o);
9          cowAS.clear();

11         cowAL.addAllAbsent(kolekce);
           cowAL.addIfAbsent(o);
13         cowAL.retainAll(kolekce);
       }
15 }
```

Concurrent Collections

- ConcurrentHashMap
  - kolekce optimalizovaná na vyhledávání prvků
  - mnohem lepší výkon v porovnání se synchronizedMap a Hashtable

Threads	*ConcurrentHashMap*	*Hashtable*
1	1,00	1,03
2	2,59	32,40
4	5,58	78,23
8	13,21	163,48
16	27,58	341,21
32	57,27	778,41

`http://www.ibm.com/developerworks/java/library/j-jtp07233.html`

  - úspěšná operace **`get()`** obvykle proběhne bez zamykání
  - na iteraci se nezamyká celá kolekce
  - mírně relaxovaná sémantika
    - při získávání prvků je možné najít i prvek, jehož vkládání ještě není dokončeno (nikdy však nesmysl)
    - iterátor může ale nemusí reflektovat změny od té doby, co byl vytvořen
    - synchronizedMap a Hashtable lze nahradit tam, kde se nespoléhá na zamykání celé tabulky

Concurrent Collections

- ConcurrentHashMap

```
1  import java.util.concurrent.ConcurrentHashMap;

3  public class CHT {
       ConcurrentHashMap cht = new ConcurrentHashMap(10);
5
       public void narabaj() {
7          cht.put(klic, objekt);
           cht.putAll(mapa);
9          cht.putIfAbsent(klic, objekt);
           cht.containsKey(klic);
11         cht.containsValue(objekt); // take contains()
           cht.entrySet();
13         cht.keySet();
           cht.values();
15         cht.clear();
       }
17 }
```

Explicitní zamykání

- potřeba jemnějšího zamykání
  - zvýšení výkonu – např. paralelizace read-only přístupů
- potřeba rozšířené funkcionality
- ReentrantLock
  - ekvivalent **`synchronized`**, pouze explicitní
  - rozšířené schopnosti (např. gettery)
  - nezapomenou správně odemknout

```
1  import java.util.concurrent.locks.ReentrantLock;

3  public class RELock {
       public static void main(String[] args) {
5          ReentrantLock relock = new ReentrantLock();
           relock.lock();
7          try {
               Thread.sleep(1000);
9              // kod
           } catch (InterruptedException e) {
11         } finally {
               relock.unlock();
13         }
       }
15 }
```

Explicitní zamykání

- ReentrantReadWriteLock
  - paralelizace na čtení, exkluzivní přístup na zápis
  - reentrantní zámek jak pro čtení, tak pro zápis
  - politiky: *writer preference* | *fair*
    specifikací v konstruktoru
  - downgrade zámku: získání read zámku před uvolněním write zámku
  - neumožňuje upgrade zámku
  - instrumentace pro monitoring (informace o držení zámků) – nikoli pro synchronizaci!
- možno si naimplementovat vlastní zámky, např. RW zámek s podporou upgrade
  - **`http://www.jtoolkit.org/articles/ReentrantReadWriteLock-upgrading.html`**
  - upgrade je nevýhodný z pohledu výkonu

Explicitní zamykání

```
1  import java.util.concurrent.locks.ReentrantReadWriteLock;

3  public class RWLock {
       boolean cacheValid = false;
5      public void pouzijCache() {
           // rwlock s fair politikou
7          ReentrantReadWriteLock rwlock = new ReentrantReadWriteLock(true);
           rwlock.readLock().lock();
9          if (!cacheValid) {
               rwlock.readLock().unlock();
11             rwlock.writeLock().lock();
               if (!cacheValid) {  // znovu zkontroluj,
13                                 // neumime upgrade bez preruseni
                   // uloz data do cache
15                 cacheValid = true;
               }
17             // rucni downgrade zamku
               rwlock.readLock().lock();  // jeste drzim na zapis
19             rwlock.writeLock().unlock();
           }
21         // pouzij data
           rwlock.readLock().unlock();
23     }
   }
```

Explicitní zamykání

- Conditions
  - čekání na splnění podmínky

```
interface Condition {
     void await() throws IE;
     boolean await(long time, TimeUnit unit) throws IE;
     long awaitNanos(long nanosTimeout) throws IE;
     void awaitUninterruptibly()
     boolean awaitUntil(Date deadline) throws IE;
     void signal();
     void signalAll();
}
```

  - výhody oproti **`wait()`**/**`notify()`**
    - více podmínek per zámek
    - absolutní a relativní timeouty
    - po návratu se dozvíme, proč jsme se vrátili
    - možnost nepřerušitelného čekání

Explicitní zamykání

```
1  import java.util.concurrent.locks.*;

3  public class OmezenyBuffer {
       Lock lock = new ReentrantLock();
5      Condition notFull = lock.newCondition();
       Condition notEmpty = lock.newCondition();
7      Object[] items = new Object[100];
       int putptr, takeptr, count;
9      public void put(Object x) throws InterruptedException {
           lock.lock();
11         try {
               while (count == items.length) notFull.await();
13             items[putptr] = x;
               if (++putptr == items.length) putptr = 0;
15             ++count;
               notEmpty.signal();
17         } finally {
               lock.unlock();
19         }
       }
21     public Object take() throws InterruptedException {
           lock.lock();
23         try {
               while (count == 0) notEmpty.await();
25             Object x = items[takeptr];
               if (++takeptr == items.length) takeptr = 0;
27             --count;
               notFull.signal();
29             return x;
           } finally {
31             lock.unlock();
           }
33     }
   }
```

http://www.cs.umd.edu/class/spring2005/cmsc433/lectures/util-concurrent-new.pdf

Executors, Thread Pools

- Koncept vykonavatelů kódu: Executors
  - vykonávají se objekty implementující Runnable
  - různé typy Executors
- ExecutorService přidává
  - schopnost zastavit vykonávání
  - schopnost vykonávat Callable<V>, nikoli pouze Runnable()
  - vracet objekty representované jako Future
- ThreadPoolExecutor
  - všeobecně použitelný executor, jednoduché API
  - minimální i maximální počet vláken
  - recyklace vláken
  - likvidace nepoužívaných vláken
  - použití si ukážeme u FutureTask

Runnable vs. Callable

- Interface Runnable
  - implementuje „střeva“ vlákna
  - lze použít s konstruktorem třídy Thread
    - konceptuálně čistější přístup: nerozšiřujeme třídu, kterou vlastně rozšiřovat nechceme
  - použití i v hlavním vlákně

```
public class PrikladRunnable {
    static class RunnableVlakno implements Runnable {
        public void run() {
            System.out.println("Tu je vlakno.");
        }
    }

    public static void main(String[] args) {
        System.out.print("Startuji vlakno: ");
        new Thread(new RunnableVlakno()).start();
        System.out.println("hotovo.");
        System.out.println("Spoustim primo v hlavnim vlakne: ");
        new RunnableVlakno().run();
    }
}
```

Runnable vs. Callable

- Interface Callable<V>
  - na rozdíl od Runnable může vracet výsledek (typu V) a vyhodit výjimku

```
1  import java.util.concurrent.Callable;

3  public class PrikladCallable {
       static class CallableVlakno implements Callable<String> {
5          public String call() throws Exception {
               return "Retezec z Callable";
7          }
       }
9
       public static void main(String[] args) {
11         try {
               String s = new CallableVlakno().call();
13             System.out.println(s);
           } catch (Exception e) {
15             System.out.println("Chytil jsem vyjimku");
           }
17     }
   }
```

Executors

- Typy Executorů
  - SingleThreadExecutor
    - sekvenční vykonávání úloh
    - pokud vlákno selže, pokračuje se vykonáváním následujícího
  - ScheduledThreadPool
    - zpožděné či opakované vykonávání vláken
  - FixedThreadPool
    - používá pevný počet vláken
  - CachedThreadPool
    - vytváří nová vlákna dle potřeby
    - opakovaně používá existující uvolněná vlákna
  - ScheduledExecutorService
    - implmentace spouštění s definovaným zpožděním a opakovaného spouštění
    - **`http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ScheduledExecutorService.html`**
  - Executors factory
    - implementace vlastních typů Executorů
    - **`http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Executors.html`**

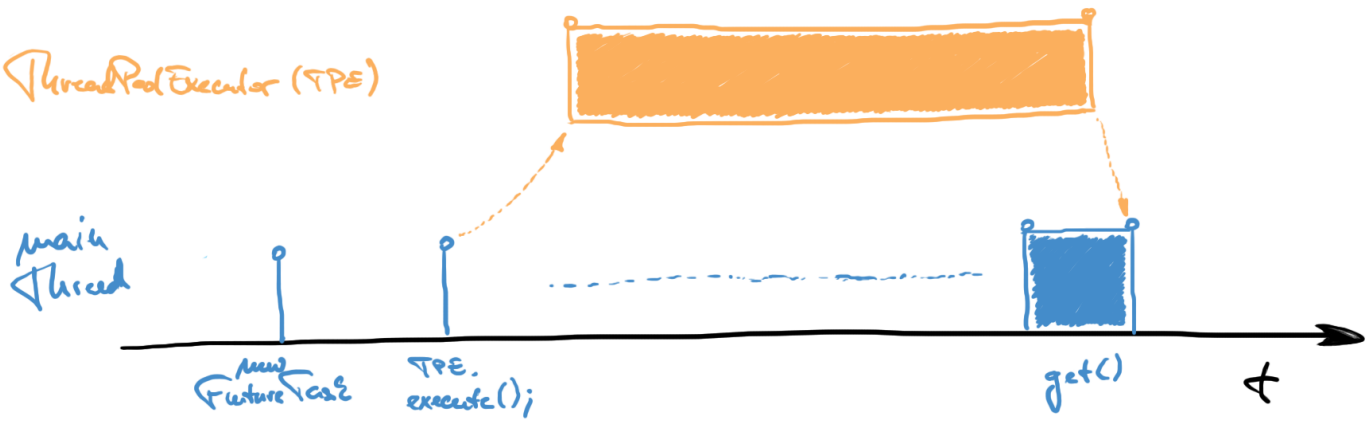
Executors

```
import java.util.concurrent.*;
import java.util.Random;

public class TPE {
    public static void main(String[] args) {
        final Random random = new Random();
        // forkbomba: ;-)
        // ExecutorService executor = Executors.newCachedThreadPool();
        ExecutorService executor = Executors.newFixedThreadPool(
                Runtime.getRuntime().availableProcessors()-1);
        for (int i = 0; i < 100; i++) {
            executor.execute(new Runnable() {
                public void run() {
                    int max = random.nextInt();
                    for(int j = 0; j < max; j++) { j += 2; j--; }
                    System.out.println("Dobehlo vlakno s max = " + max);
                }
            });
        }
        try {
            Thread.sleep(10000);
            executor.shutdown();
            executor.awaitTermination(1000, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
        }
    }
}
```

Futures

- Princip:
  - někdy v budoucnu bude volající potřebovat výsledek výpočtu *X*
  - v době, kdy si volající řekne o výsledek výpočtu *X*: *(a)* výsledek je okamžitě vrácen, pokud je již k dispozici, nebo *(b)* volající se zablokuje, výsledek se dopočítá a vrátí, volající se odblokuje
- H. Baker, C. Hewitt, "The Incremental Garbage Collection of Processes". *Proceedings of the Symposium on Artificial Intelligence Programming Languages, SIGPLAN Notices 12.* August 1977.
  podobný koncept
  D. Friedman. "CONS should not evaluate its arguments". S. Michaelson and R. Milner, editors, *Automata, Languages and Programming*, pages 257-284. Edinburgh University Press, Edinburgh. Also available as *Indiana University Department of Computer Science Technical Report TR44*. 1976



Futures a ThreadPoolExecutor

```
1  import java.util.concurrent.*;

3  public class Futures {
       public static class StringCallable implements Callable {
5          public String call() throws Exception {
               System.out.println("FT: Pocitam.");
7              Thread.sleep(5000);
               System.out.println("FT: Vypocet hotov.");
9              return "12345";
           }
11     }
       public static void main(String[] args) {
13         ThreadPoolExecutor tpe = new ThreadPoolExecutor(2, 8, 60L,
                   TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
15         FutureTask ft = new FutureTask(new StringCallable());
           System.out.println("main: Poustim vypocet.");
17         tpe.execute(ft);
           // alternativa: Future ft = tpe.submit(new StringCallable());
19         try {
               System.out.println("main: Chci vysledek.");
21             String s = (String) ft.get();
               System.out.println("main: Mam vysledek: " + s);
23             tpe.shutdown();
               tpe.awaitTermination(1, TimeUnit.MINUTES);
25         } catch (InterruptedException e) {}
           catch (ExecutionException e) {}
27     }
   }
```

Programování v reálném čase

- **`http://www.rtsj.org/`**
- P. Dibble: Real-Time Java Platform Programming
  `http://www.sun.com/books/catalog/dibble.xml`
  - Interoperability with non-RT code, tradeoffs in real-time development, and RT issues for the JVM software
  - Garbage collection, non-heap access, physical and „immortal“ memory, and constant-time allocation of non-heap memory
  - Priority scheduling, deadline scheduling, and rate monotonic analysis
  - Closures, asynchronous transfer of control, asynchronous events, and timers