Vláknové programování
část III

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

Laboratoř pokročilých síťových technologií

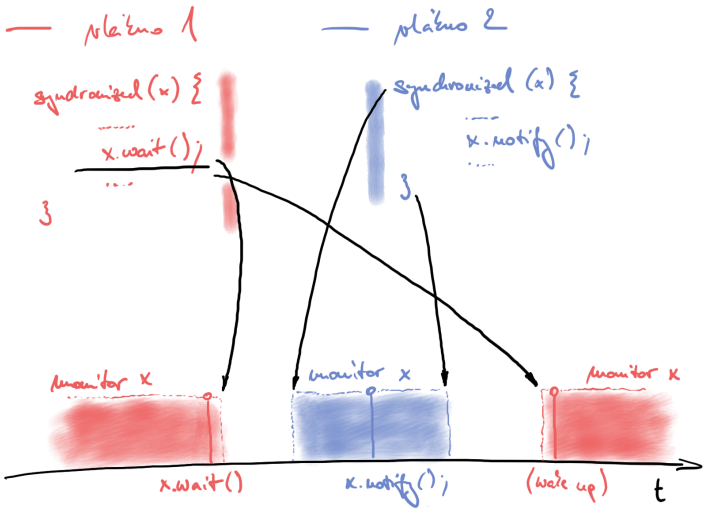
PV192
2012–03–13

Přehled přednášky

Signalizace mezi objekty

- Definováno pro každý Object
- Musí být vlastníkem monitoru pro daný Objekt
  - `synchronized` sekce
- Metoda `wait()`
  - usnutí do doby notifikace
  - při usnutí se vlákno vzdá monitoru
  - po probuzení čeká, než monitor může opět získat
- Metoda `notify()`
  - notifikace jednoho z čekajících
  - pokud je čekajících více, vybere se jeden (libovolně dle implementace)
  - vybuzené vlákno pokračuje až poté, co se notifikující vlákno vzdá monitoru
- Metoda `notifyAll()`
  - notifikace všech vláken čekajících na daném objektu

Signalizace mezi objekty



Suspendování vláken

- Metody `Thread.suspend()` a `Thread.resume` jsou inherentně nebezpečné – deadlocky
- Emulace pomocí `wait()` a `notify()`

Suspendování vláken

```
1  import static java.lang.Thread.sleep;

3  public class PrikladSuspendu {
       static class MojeVlakno extends Thread {
5          private volatile boolean ukonciSe = false;
           private volatile boolean spi = false;
7
           public void run() {
9              while (!ukonciSe) {
                   System.out.println("...makam...");
11                 try {
                       sleep(500);
13                     synchronized (this) {
                           while (spi) {
15                             wait();
                           }
17                     }
                   } catch (InterruptedException e) {
19                     System.out.println("Necekane probuzeni!");
                   }
21             }
               System.out.println("...domakal jsem...");
23         }
```

Suspendování vláken

```
1         public void skonci() {
              ukonciSe = true;
3         }

5         public void usni() {
              spi = true;
7         }

9         public void vzbudSe() {
              spi = false;
11            synchronized (this) {
                  this.notify();
13            }
          }
15    }
```

Suspendování vláken

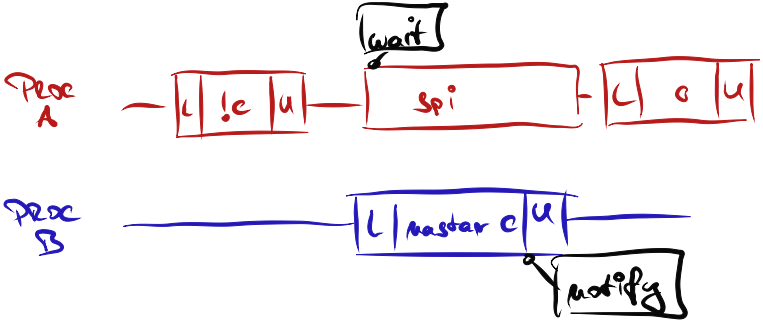
- Ztracené zprávy
  - `o.wait()` a `o.notify()` resp. `o.notifyAll` nemají mechanismus zdržení notifikace
  - pokud vlákno usne na `o.wait()` později, než mělo být notifikováno přes `o.notify`, nikdy se nevzbudí

$\Longrightarrow$ deadlock

- Problém při signalizaci s podmínkami
  - odpovídá Hoareho monitorům
  - vlákno usne do doby, než je splněna podmínka
  - v době vzbuzení je garantováno, že je podmínka pořád splněna
  - implementace Hoarových monitorů pro Javu:
    `http://www.engr.mun.ca/%7Etheo/Misc/monitors/monitors.html`
    `http://www.javaworld.com/javaworld/jw-10-2007/jw-10-monitors.html`

Suspendování vláken

- Podmíněná signalizace



Suspendování vláken

```
        // podmineny predikat musi byt chraneny zamkem
2       synchronized (lock) {
            while (!conditionPredicate)
4               lock.wait();
            // nyni je objekt v pozadovanem stavu
6       }
```

- Pravidla pro signalizaci s podmínkami
  1. zformulovat a ověřit podmínku před voláním `wait()`
  2. `wait()` bežet ve smyčce, kontrolovat po vzbuzení
     - probuzení z `wait()` mohlo nastat z jiného důvodu
  3. zajistit, aby proměnné v podmínce byly chráněny tím zámkem, který se používá v monitoru
  4. držet zámek v době volání `wait()`, `notify()`, `notifyAll()`
- Potřeba zajistit, aby při **změně** podmínky vždy někdo zasignalizoval
- Signál se může ztratit, pokud bychom se vzdali mezi dalším testem monitoru

Vzor producent–konzument

- Třídy Queue a BlockingQueue
  - metody:
    - `offer()` přidává na konec fronty (blokuje se v případě BlockingQueue a zaplnění kapacity)
    - nepoužívat `add()` pro fronty s omezenou kapacitou
    - `peek()` vrátí prvek ze začátku fronty, ale neodstraní ho z fronty
    - `poll()` vrátí prvek ze začátku fronty, `null` pokud je fronta prázdná
    - `remove()` vrátí prvek ze začátku fronty, výjimka NoSuchElementException pokud je fronta prázdná
    - `take()` vrátí prvek ze začátku blokující fronty, nebo se zablokuje, dokud je fronta prázdná
  - typy
    - ConcurrentLinkedQueue – neblokující, FIFO, efektivní wait-free algoritmus, nesmí obsahovat `null`
    - PriorityQueue – podpora priority (přirozené uspořádání, `public interface Comparable<T>`)
    - LinkedBlockingQueue – blokující obdoba ConcurrentLinkedQueue
    - PriorityBlockingQueue – blokující obdoba PriorityQueue
    - SynchronousQueue – synchronní blokující fronta (`offer()` se zablokuje až do odpovídajícího `take()`)

Vzor producent–konzument

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

public class Fronty {
    public class NeblokujiciFronty {
        Queue clq = new ConcurrentLinkedQueue();
        Queue pq = new PriorityQueue(50);
        Queue q = new SynchronousQueue();

    }

    public class BlokujiciFronty {
        BlockingQueue bclq = new LinkedBlockingQueue(30);
        BlockingQueue bpq = new PriorityBlockingQueue();

        void pouziti() {
            bclq.offer(new Object());
            Object o = bclq.peek();
            o = bclq.poll();
            try {
                o = bclq.take();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

}
```

Vzor producent–konzument

- Vzor producent–konzument
  - producenti přidávají práci do fronty (`offer()`)
  - konzumenti přidávají práci do fronty (`take()`)
  - zvláště zajímavé se thread pools

Vzor producent–konzument

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

public class ProducentKonzument extends Thread {
    public class Task {
    }
    BlockingQueue<Task> bclq = new LinkedBlockingQueue<Task>();

    public void run() {
        Thread producent = new Thread() {
            public void run() {
                bclq.offer(new Task());
            }
        };

        Thread konzument = new Thread() {
            public void run() {
                try {
                    Task t = bclq.take();
                } catch (InterruptedException e) {
                    System.out.println("Necekane probuzeni!");
                }
            }
        };

        producent.start();
        konzument.start();
    }
}
```

Vzor kradení práce

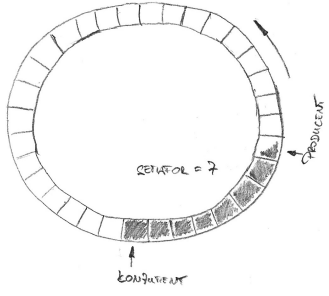
- Deque a BlockingDeque
  - umožňují vybírat prvky ze začátku i z konce fronty
  - normální konzumenti vybírají prvky ze začátku fronty
  - vlákna, která se „nudí“ mohou převzít práci z konce fronty
  - např. udržování fronty per vlákno, „nudící se“ vlákna mohou koukat do cizích front
  - vhodné např. pro situace, kdy si vlákno generuje další práci samo pro sebe (webový crawler)

Vzor kradení práce

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

public class KradeniPrace {
    public class Task {
    }
    BlockingDeque<Task> deque = new LinkedBlockingDeque<Task>(20);

    public void run() {
        Thread producent = new Thread() {
            public void run() { deque.offer(new Task()); }
        };

        Thread konzument1 = new Thread() {
            public void run() {
                try {
                    Task t = deque.take();
                } catch (InterruptedException e) {
                }
            }
        };

        Thread konzument2 = new Thread() {
            public void run() { Task t = deque.pollLast(); }
        };

        producent.start(); konzument1.start(); konzument2.start();
    }
}
```

Další synchronizační prvky

- semafory
  - počáteční kapacita *N* „permitů“
  - **`acquire()`** získá „permit“, eventuálně se zablokuje, pokud permity došly
  - **`release()`** vrátí permit



Další synchronizační prvky

- závlačka – CountDownLatch
  - speciální typ semaforu, z jehož kapacity lze jen odečítat
  - **`await()`** čeká, až hodnota klesne na 0
  - např. čekání na až doběhne *n* nějakých událostí

```
import java.util.concurrent.CountDownLatch;

public class Zavlacka extends Thread {
    static final int POCET_UDALOSTI = 10;
    CountDownLatch cdl = new CountDownLatch(POCET_UDALOSTI);
    public void run() {
        Thread ridici = new Thread(){
            public void run() {
                for (int i = 0; i < POCET_UDALOSTI; i++) {
                    cdl.countDown();
                }
            }
        };
```

Další synchronizační prvky

- závlačka – CountDownLatch

```
14          Thread cekaci  = new Thread() {
                public void run() {
16                  try {
                        System.out.println("Musim pockat na "
18                              + POCET_UDALOSTI + " udalosti");
                        cdl.await();
20                      System.out.println("Ted teprv muzu bezet.");
                    } catch (InterruptedException e) {
22                      System.out.println("Neocekavane vzbuzeni!");
                    }
24              }
            };
26          cekaci.start(); ridici.start();
        }
28
        public static void main(String[] args) {
30          new Zavlacka().start();
        }
32  }
```

Další synchronizační prvky

- FutureTask
  - podrobně si koncept probereme u Futures a ThreadPoolExecutors
  - je implementována pomocí Callable
    - obdoba Runnable, akorát umožňuje vracet hodnotu
  - metoda `get()` umožňuje čekat, než je k dispozici návratová hodnota

Další synchronizační prvky

- bariéry
  - umožňuje více vláknům se se jít v jednom místě
  - např. pro iterativní výpočty, kde jedna iterace může být rozdělena na *n* paralelních a další iterace je závislá na výsledku předchozí iterace
  - zatímco závlačky jsou určeny k čekání na události, bariéry jsou určeny k čekání na jiná vlákna
  - CyclicBarrier – bariéra pro opakované setkávání se konstantního počtu vláken
  - pokud se nějaké vlákno vzbudí během `await()` metody, považuje se bariéra za prolomenou a všichni ostatní čekající dostanou BrokenBarrierException
- Exchanger
  - výměna dat během bariéry
  - ekvivalent konceptu rendezvous v Adě

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

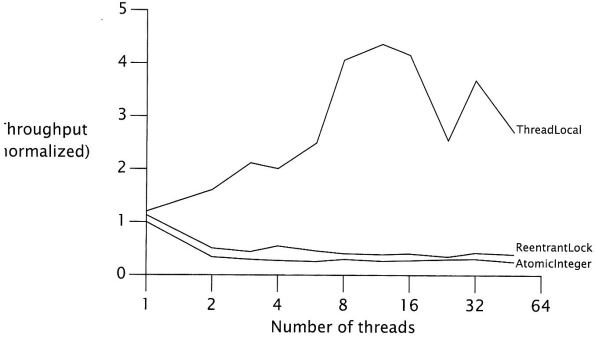


FIGURE 15.1. Lock and `AtomicInteger` performance under high contention.

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

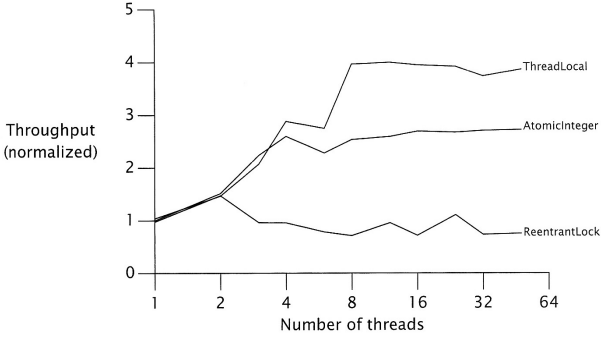


FIGURE 15.2. `Lock` and `AtomicInteger` performance under moderate contention.

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

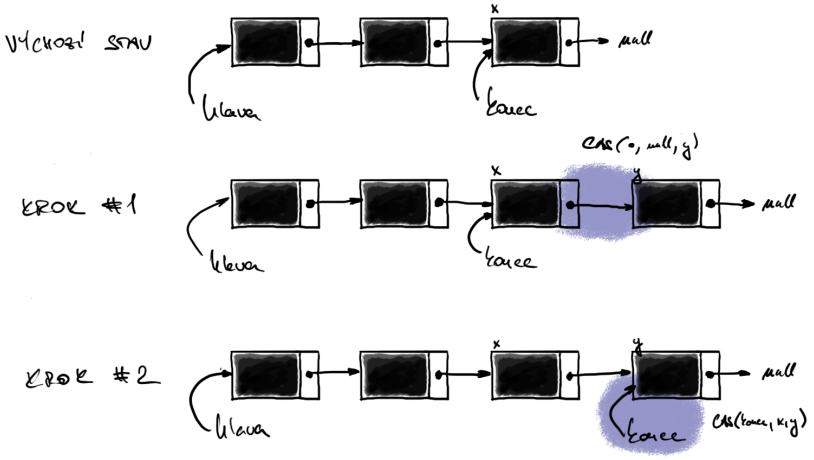
Atomické typy

- `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

```
import java.util.concurrent.atomic.AtomicReference;
// dle http://www.javaconcurrencyinpractice.com/listings/LinkedQueue.java

public class AtomickySeznam<E> {
    private static class Node<E> {
        final E polozka;
        final AtomicReference<AtomickySeznam.Node<E>> next;

        public Node(E polozka, AtomickySeznam.Node<E> next) {
            this.polozka = polozka;
            this.next = new
                    AtomicReference<AtomickySeznam.Node<E>>(next);
        }
    }

    private final AtomickySeznam.Node<E> dummy =
            new AtomickySeznam.Node<E>(null, null);
    private final AtomicReference<AtomickySeznam.Node<E>> hlava
            = new AtomicReference<AtomickySeznam.Node<E>>(dummy);
    private final AtomicReference<Node<E>> konec
            = 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.*;
2
3  public class CoW {
4      CopyOnWriteArraySet cowAS = new CopyOnWriteArraySet();
5      CopyOnWriteArrayList cowAL = new CopyOnWriteArrayList();
6      public void narabaj() {
7          cowAS.addAll(kolekce);
8          cowAS.contains(o);
9          cowAS.clear();
10
11         cowAL.addAllAbsent(kolekce);
12         cowAL.addIfAbsent(o);
13         cowAL.retainAll(kolekce);
14     }
15 }
```

Concurrent Collections

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

Threads	*ConcurrentHashMap [ms/10 Mops]*	*Hashtable [ms/10 Mops]*
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

`https://www.ibm.com/developerworks/java/library/j-jtp07233/index.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

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

public class CHT {
    ConcurrentHashMap cht = new ConcurrentHashMap(10);

    public void narabaj() {
        cht.put(klic, objekt);
        cht.putAll(mapa);
        cht.putIfAbsent(klic, objekt);
        cht.containsKey(klic);
        cht.containsValue(objekt); // take contains()
        cht.entrySet();
        cht.keySet();
        cht.values();
        cht.clear();
    }
}
```

Explicitní zamykání

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

Explicitní zamykání

- Conditions
  - výhody oproti `wait()`/`notify()`
    - více podmínek per zámek

```
final Lock zamek = new ReentrantLock();
final Condition nePlny  = zamek.newCondition();
final Condition nePrazdny = zamek.newCondition();
```

    - absolutní a relativní timeouty
    - po návratu se dozvíme, proč jsme se vrátili
    - možnost nepřerušitelného čekání (nereaguje na metodu `interrupt`)
  - může se vyskytnout *spurious wakeup*
    - je třeba používat idiom ověřování stavu podmínky! :(

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

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