

# Vláknové programování

## část IV

**Lukáš Hejtmánek, Petr Holub**

`{xhejtman, hopet}@ics.muni.cz`



Laboratoř pokročilých síťových technologií

PV192  
2012-03-13

# Přehled přednášky

Monitory a synchronizace

Signalizace a suspend

Paralelní vzory

Pokročilé vlastnosti Javy

Atomické typy

Concurrent Collections

Explicitní zamykání

# synchronized a monitor

- Monitorování – Hoare, Dijkstra, Hansen, cca 1974
  - vynucuje serializovaný přístup k objektu
- `synchronized` – základní nástroj pro vyloučení v kritické sekci
  - v Javě se označuje jako monitor
  - synchronizuje se na explicitně uvedeném objektu (raději `final`) nebo (implicitně) na `this`
  - Javové monitory nezahrnují podmínky, jsou jednodušší než Hoareho

## synchronized a monitory

```
1 import net.jcip.annotations.ThreadSafe;
   @ThreadSafe
3 public class PříkladSynchronized {
       Integer cislo;
5     public PříkladSynchronized() {
           this.cislo = 0;
7     }
       public PříkladSynchronized(Integer cislo) {
9           this.cislo = cislo;
       }
11    void pricti(int i) {
           synchronized (this) {
13           cislo += i;
           }
15    }
       synchronized int kolikJeCislo() {
17           return cislo;
       }
19 }
```

# synchronized a monitorování

- *Java monitor pattern*

```
1 import net.jcip.annotations.GuardedBy;
  // http://www.javaconcurrencyinpractice.com/jcip-annotations.jar
3
4 public class MonitorPattern {
5     private final Object zamek = new Object();
6     @GuardedBy("zamek") Object mujObject;
7
8     void metoda() {
9         synchronized (zamek) {
10             // manipulace s objektem mujObject;
11         }
12     }
13 }
```

# Synchronized Collections

- Přímě synchronizované kolekce:  
**Vector, Hashtable**
- Synchronizované obaly:  
**Collection.synchronizedX**  
factory metody
- Tyto kolekce jsou thread-safe, ale poněkud zákeřné
  - může být potřeba chránit pomocí zámků složené akce
    - ◆ iterace
    - ◆ navigace (procházení prvků v nějakém pořadí)
    - ◆ podmíněné operace, např. vlož-pokud-chybí (put-if-absent)

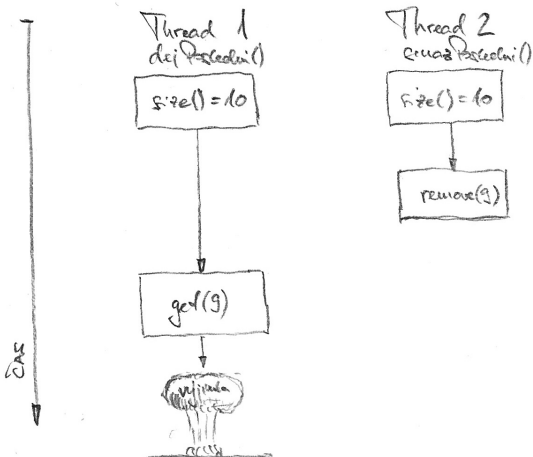
# Synchronized Collections

```
1 import java.util.Vector;
3 public class PodlaKolekce {
4     public static Object dejPosledni(Vector v) {
5         int posledni = v.size() - 1;
6         return v.get(posledni);
7     }
9     public static void smazPosledni(Vector v) {
10        int posledni = v.size() - 1;
11        v.remove(posledni);
12    }
13 }
```



# Synchronized Collections

PROBLÉMY SYNCHRONIZOVANÉ KOLEKCE





# Synchronized Collections

- Výše uvedené chování nemůže poškodit `vector v`  $\implies$  thread-safe
- Chování ale bude zmatečné
  - mezi získání indexu poslední položky a `get ()` ev. `remove ()` se může vloučit jiný `remove ()`  $\implies$  vyhazování výjimky `ArrayOutOfBoundsException`
- Lze ošetřit klientským zamykáním, pokud známe objekt, na němž se v synchronizované kolekci dělá monitor

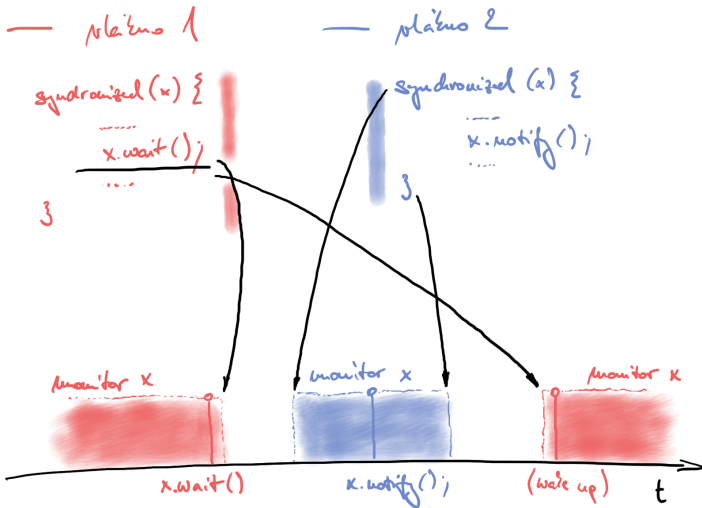
# Synchronized Collections

```
1 import java.util.Vector;
3 public class RucneSynchnutaKolekce {
4     public static Object dejPosledni(Vector v) {
5         synchronized (v) {
6             int posledni = v.size() - 1;
7             return v.get(posledni);
8         }
9     }
11    public static void smazPosledni(Vector v) {
12        synchronized (v) {
13            int posledni = v.size() - 1;
14            v.remove(posledni);
15        }
16    }
17 }
```

# 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

```
import static java.lang.Thread.sleep;
2
public class PrikladSuspendu {
4     static class MojeVlakno extends Thread {
        private volatile boolean ukonciSe = false;
6         private volatile boolean spi = false;

8         public void run() {
            while (!ukonciSe) {
10                System.out.println("...makam...");
                try {
12                    sleep(500);
                    synchronized (this) {
14                        while (spi) {
                            wait();
16                        }
                    }
18                } catch (InterruptedException e) {
                    System.out.println("Necekane probuzeni!");
20                }
            }
22            System.out.println("...domakal jsem...");
        }
    }
}
```

# Suspendování vláken

```
1      public void skonci() {
2          ukonciSe = true;
3      }
4
5      public void usni() {
6          spi = true;
7      }
8
9      public void vzbudSe() {
10         spi = false;
11         synchronized (this) {
12             this.notify();
13         }
14     }
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

⇒ **deadlock**

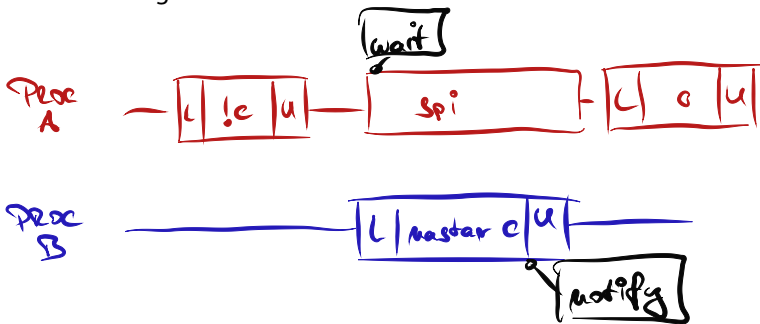
- Problém při signalizaci s podmínkami

- 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

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

- Pravidla pro signalizaci s podmínkami
  1. zformulovat a ověřit podmínku před voláním `wait ()`
  2. `wait ()` běžet 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 zámeček v době volání `wait ()`, `notify ()`, `notifyAll ()`
- Potřeba zajistit, aby při **změně** podmínky vždy někdo zsignalizoval
- 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

```
2 import java.util.*;
3 import java.util.concurrent.*;
4 public class Fronty {
5     public class NeblokujiciFronty {
6         Queue clq = new ConcurrentLinkedQueue();
7         Queue pq = new PriorityQueue(50);
8         Queue q = new SynchronousQueue();
9     }
10
11     public class BlokujiciFronty {
12         BlockingQueue bclq = new LinkedBlockingQueue(30);
13         BlockingQueue bpq = new PriorityBlockingQueue();
14
15         void pouziti() {
16             bclq.offer(new Object());
17             Object o = bclq.peek();
18             o = bclq.poll();
19             try {
20                 o = bclq.take();
21             } catch (InterruptedException e) {
22                 e.printStackTrace();
23             }
24         }
25     }
26 }
28 }
```

# 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.*;
2
public class ProducentKonzument extends Thread {
4     public class Task {
        }
6     BlockingQueue<Task> bclq = new LinkedBlockingQueue<Task> ();

8     public void run() {
        Thread producent = new Thread() {
10         public void run() {
            bclq.offer(new Task());
12         }
        };

14         Thread konzument = new Thread() {
16         public void run() {
            try {
18                 Task t = bclq.take();
                } catch (InterruptedException e) {
20                 System.out.println("Necekane probuzeni!");
                }
22         }
        };

24         producent.start ();
26         konzument.start ();
        }
28 }

```

# 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.*;
2
public class KradeniPrace {
4     public class Task {
        }
6     BlockingDeque<Task> deque = new LinkedBlockingDeque<Task>(20);

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

12         Thread konzument1 = new Thread() {
14         public void run() {
            try {
16                 Task t = deque.take();
                } catch (InterruptedException e) {
18                 }
            }
20         };

22         Thread konzument2 = new Thread() {
            public void run() { Task t = deque.pollLast(); }
24         };

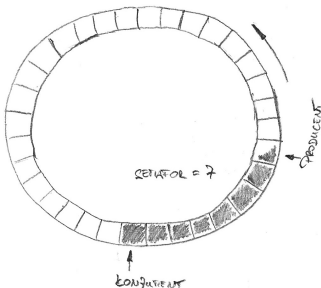
26         producent.start(); konzument1.start(); konzument2.start();
        }
28 }
```



## Další synchronizační prvky

- semaforey

- 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;
2
public class Zavlačka extends Thread {
4     static final int POCET_UDALOSTI = 10;
    CountDownLatch cdl = new CountDownLatch(POCET_UDALOSTI);
6     public void run() {
        Thread ridici = new Thread(){
8         public void run() {
            for (int i = 0; i < POCET_UDALOSTI; i++) {
10                cdl.countDown();
            }
12        }
    };
};
```

## Další synchronizační prvky

- závlačka – CountdownLatch

```
14     Thread cekaci = new Thread() {
15         public void run() {
16             try {
17                 System.out.println("Musim pockat na "
18                     + PO CET_UDALOSTI + " udalosti");
19                 cdl.await();
20                 System.out.println("Ted teprv muзу bezet.");
21             } catch (InterruptedException e) {
22                 System.out.println("Neocekavane vzbuzeni!");
23             }
24         }
25     };
26     cekaci.start(); ridici.start();
27 }
28
29 public static void main(String[] args) {
30     new Zavlacka().start();
31 }
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 vlákn
- atomické proměnné to zvládnou bez přepínání kontextu
  - vyšší výkon pro nízkou až střední míru soutěžení o zámek (lock contention)
  - tzv. wait-free synchronizace

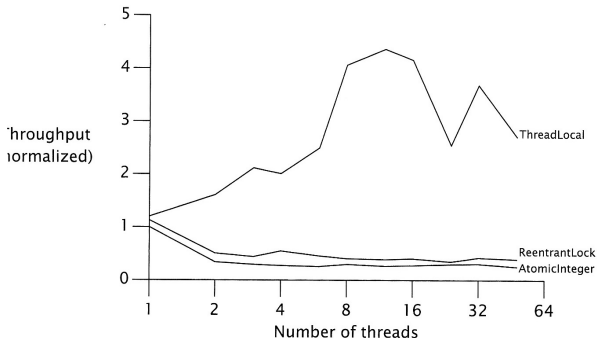


FIGURE 15.1. Lock and AtomicInteger performance under high contention.

# Atomické typy

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

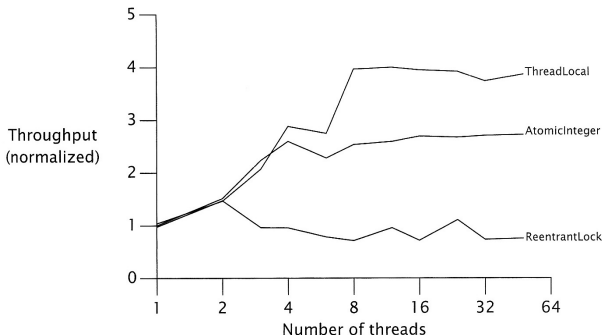


FIGURE 15.2. Lock and AtomicInteger performance under moderate contention.

# 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  $\implies$  (pomalá) podpora u Motorola 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 lthania



# 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_1/stl_c` a `ldq_1/stq_c` (Alpha), `lwarx/stwax` (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 vrátí původní hodnotu

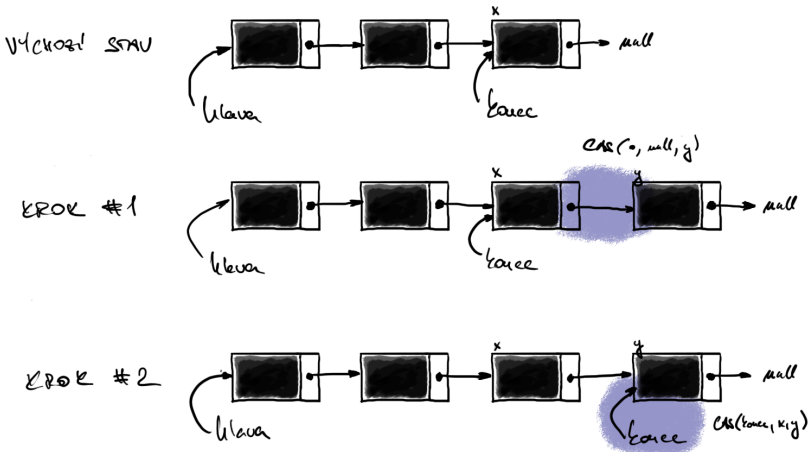
# Atomické typy

- **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;
2 // dle http://www.javaconcurrencyinpractice.com/listings/LinkedList.java

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

8         public Node(E polozka, AtomickySeznam.Node<E> next) {
10             this.polozka = polozka;
                this.next = new
12                 AtomicReference<AtomickySeznam.Node<E>>(next);
            }
14     }

16     private final AtomickySeznam.Node<E> dummy =
        new AtomickySeznam.Node<E>(null, null);
18     private final AtomicReference<AtomickySeznam.Node<E>> hlava
        = new AtomicReference<AtomickySeznam.Node<E>>(dummy);
20     private final AtomicReference<Node<E>> konec
        = new AtomicReference<AtomickySeznam.Node<E>>(dummy);
```

## Neblokující seznam: Michael-Scott, 1996

```

22 public boolean put(E položka) {
24     AtomickySeznam.Node<E> newNode =
26         new AtomickySeznam.Node<E>(položka, null);
28     while (true) {
30         AtomickySeznam.Node<E> curkonec = konec.get();
32         AtomickySeznam.Node<E> konecNext = curkonec.next.get();
34         if (curkonec == konec.get()) {
36             if (konecNext != null) {
38                 // dokoncime rozpracovany stav - posuneme konec
40                 konec.compareAndSet(curkonec, konecNext);
42             } else {
44                 // pokusime se vlozit
46                 if (curkonec.next.compareAndSet(null, newNode)) {
48                     // pri uspechu se pokusime posunout konec
50                     konec.compareAndSet(curkonec, newNode);
52                     return true;
54                 }
56             }
58         }
60     }
62 }

```

# 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 {
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

<i>Threads</i>	<i>ConcurrentHashMap [ms/10 Mops]</i>	<i>Hashtable [ms/10 Mops]</i>
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 nespolehá na zamykání celé tabulky

# Concurrent Collections

- ConcurrentHashMap

```
1 import java.util.concurrent.ConcurrentHashMap;
3 public class CHT {
4     ConcurrentHashMap cht = new ConcurrentHashMap(10);
5
6     public void narabaj() {
7         cht.put(klic, objekt);
8         cht.putAll(mapa);
9         cht.putIfAbsent(klic, objekt);
10        cht.containsKey(klic);
11        cht.containsValue(objekt); // take contains()
12        cht.entrySet();
13        cht.keySet();
14        cht.values();
15        cht.clear();
16    }
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 {
4     public static void main(String[] args) {
5         ReentrantLock relock = new ReentrantLock();
6         relock.lock();
7         try {
8             Thread.sleep(1000);
9             // kod
10        } catch (InterruptedException e) {
11        } finally {
12            relock.unlock();
13        }
14    }
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 {
4     boolean cacheValid = false;
5     public void pouzijCache() {
6         // rwlock s fair politikou
7         ReentrantReadWriteLock rwlock = new ReentrantReadWriteLock(true);
8         rwlock.readLock().lock();
9         if (!cacheValid) {
10            rwlock.readLock().unlock();
11            rwlock.writeLock().lock();
12            if (!cacheValid) { // znovu zkontroluj,
13                               // neumime upgrade bez preruseni
14                // uloz data do cache
15                cacheValid = true;
16            }
17            // rucni downgrade zamku
18            rwlock.readLock().lock(); // jeste drzim na zapis
19            rwlock.writeLock().unlock();
20        }
21        // pouzij data
22        rwlock.readLock().unlock();
23    }
24 }

```

# 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 nePrasny = 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 {
4      Lock lock = new ReentrantLock();
5      Condition notFull = lock.newCondition();
6      Condition notEmpty = lock.newCondition();
7      Object[] items = new Object[100];
8      int putptr, takeptr, count;
9      public void put(Object x) throws InterruptedException {
10         lock.lock();
11         try {
12             while (count == items.length) notFull.await();
13             items[putptr] = x;
14             if (++putptr == items.length) putptr = 0;
15             ++count;
16             notEmpty.signal();
17         } finally {
18             lock.unlock();
19         }
20     }
21     public Object take() throws InterruptedException {
22         lock.lock();
23         try {
24             while (count == 0) notEmpty.await();
25             Object x = items[takeptr];
26             if (++takeptr == items.length) takeptr = 0;
27             --count;
28             notFull.signal();
29             return x;
30         } finally {
31             lock.unlock();
32         }
33     }
34 }

```



# 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