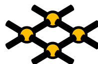
Vláknové programování
část III

Lukáš Hejmánek, Petr Holub

{xhejtman,hopet}@ics.muni.cz



Laboratoř pokročilých síťových technologií

PV192
2013–03–04

Přehled přednášky

Vlákna v jazyce Java

Paralelní vzory

Pokročilé vlasnosti Javy
- Atomické typy
- Concurrent Collections
- Explicitní zamykání

Měření

Přehled přednášky

Vlákna v jazyce Java

Paralelní vzory

Pokročilé vlasnosti Javy
- Atomické typy
- Concurrent Collections
- Explicitní zamykání

Měření

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

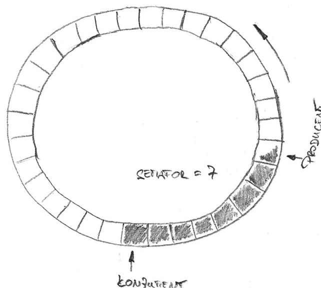
- Deque a BlockingDeque
  - umožňují vybírat prvky ze začátku i z konce fronty
  - normální konzumenti vybírají prvky ze začátku fronty
  - vlákna, která se „nudí“ mohou převzít práci z konce fronty
  - např. udržování fronty per vlákno, „nudící se“ vlákna mohou koukat do cizích front
  - vhodné např. 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

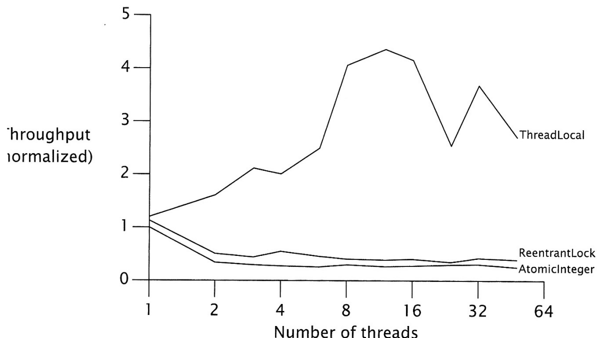


FIGURE 15.1. Lock and AtomicInteger performance under high contention.

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

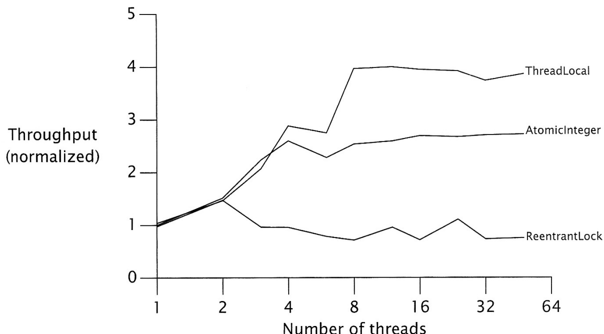


FIGURE 15.2. Lock and `AtomicInteger` performance under moderate contention.

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

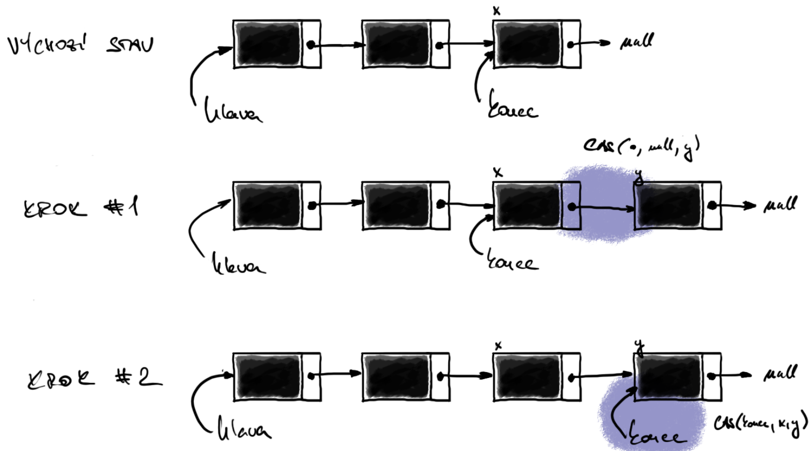
Atomické typy

- **`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 $A \to B \to A$
  - 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

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

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

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

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

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

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

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

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

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

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

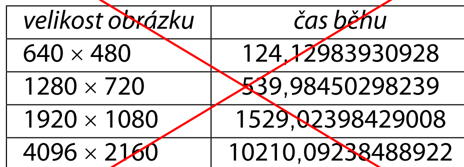
- **`http://www.rtsj.org/`**
- P. Dibble: Real-Time Java Platform Programming
  `http://www.sun.com/books/catalog/dibble.xml`
  - Interoperability with non-RT code, tradeoffs in real-time development, and RT issues for the JVM software
  - Garbage collection, non-heap access, physical and "immortal"memory, 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

Interakce s JVM při měření

- Problém garbage collection
  - `-verbose:gc`
  - krátká měření: vybrat pouze běhy, v nichž nedošlo ke GC
  - dlouhé běhy: dostatečně dlouhé, aby se přítomnost GC projevila representativně
- Problém HotSpot kompilace
  - `-XX:+PrintCompilation`
  - dostatečný warm-up (minuty!)
  - mohou se vyskytovat rekompilace (optimalizace, nahrání nové třídy která zruší dosavadní předpoklady)
  - housekeeping tasks: oddělení nesouvisejících měření pauzou nebo restartem JVM

Délka zpracování obrázku

velikost obrázku	*čas běhu*
640 × 480	124,12983930928
1280 × 720	539,98450298239
1920 × 1080	1529,02398429008
4096 × 2160	10210,09238488922



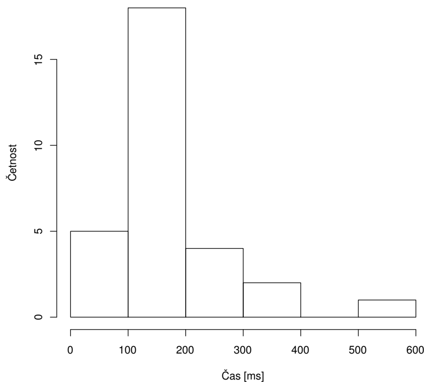
velikost obrázku	*čas běhu*
640 × 480	124,12983930928
1280 × 720	539,98450298239
1920 × 1080	1529,02398429008
4096 × 2160	10210,09238488922

Měříme délku výpočtu v Javě

```
> library(psych)
> runlength <- read.csv(file="java-example.table", head=FALSE, sep=","
> summary(runlength$V1)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  92.08  104.70  108.80  166.80  187.20  594.70
> describe(runlength$V1)
  var  n   mean     sd median trimmed   mad   min    max  range skew k
1   1 30 166.82 113.67 108.78   142.1 20.88 92.08 594.71 502.63 2.14
4.55
     se
1 20.75
```

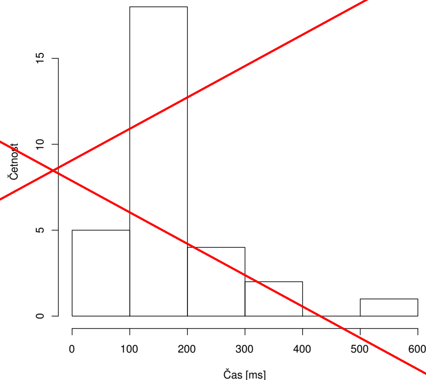
$$N = 30$$
$$\overline{x} = 166{,}82$$
$$sx = 113{,}67$$
$$s_{\overline{x}} = \frac{sx}{\sqrt{N}} = 20{,}75$$
$$t_{0{,}05;29} = 2{,}045$$

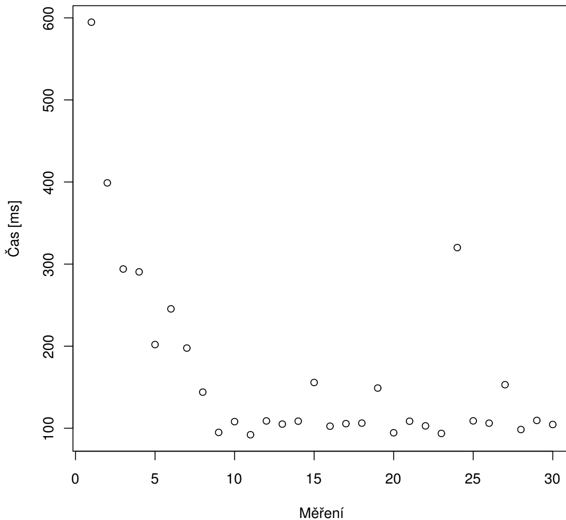
$$\overline{x} \pm t_{0{,}05;N-1} s_{\overline{x}} = 167 \pm 42\,\text{ms}$$

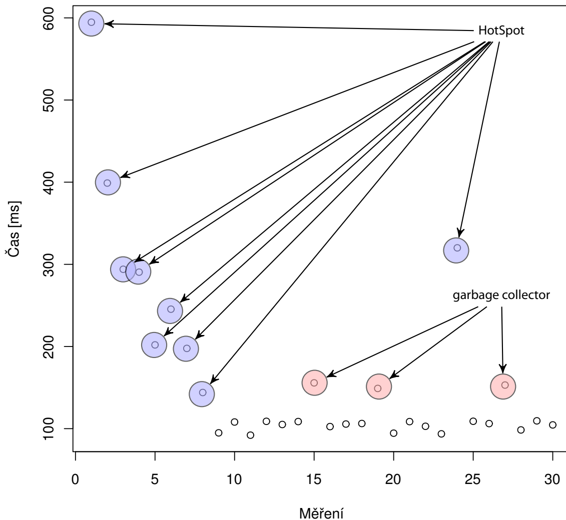


$$N = 30$$
$$\overline{x} = 166{,}82$$
$$sx = 113{,}67$$
$$s_{\overline{x}} = \frac{sx}{\sqrt{N}} = 20{,}75$$
$$t_{0{,}05;29} = 2{,}045$$

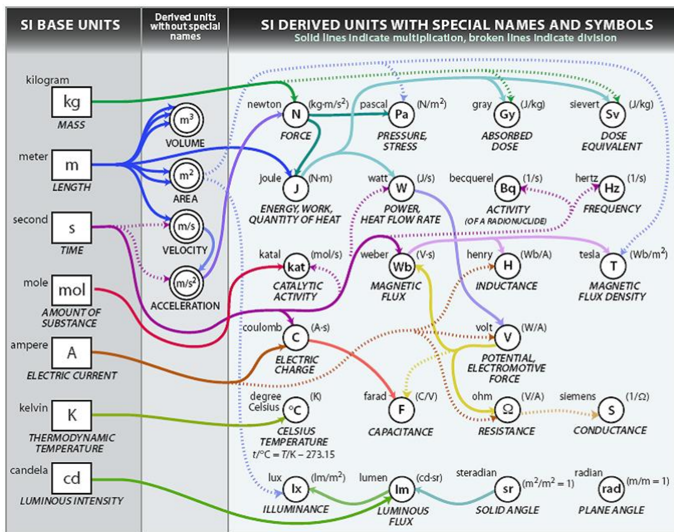
$$\overline{x} \pm t_{0{,}05;N-1} s_{\overline{x}} = 167 \pm 42\,\text{ms}$$




Javové měření
Čas [ms]
600
500
400
300
200
100
0
5
10
15
20
25
30
Měření


Javové měření
HotSpot
garbage collector
Čas [ms]
Měření
100
200
300
400
500
600
0
5
10
15
20
25
30

Soustava jednotek pro informatiky



Zdroj: `http://www.icrf.nl/Portals/106/SI_units_diagram(1).jpg`

Soustava jednotek pro informatiky

- Předpony nejen speciálně informatické

yocto-	10^{-24}	y	–	–	–
zepto-	10^{-21}	z	–	–	–
atto-	10^{-18}	a	–	–	–
femto-	10^{-15}	f	–	–	–
pico-	10^{-12}	p	–	–	–
nano-	10^{-9}	n	–	–	–
micro-	10^{-6}	μ	–	–	–
milli-	10^{-3}	m	–	–	–
kilo-	10^{3}	k	kibi	2^{10}	Ki
mega-	10^{6}	M	mebi	2^{20}	Mi
giga-	10^{9}	G	gibi	2^{30}	Gi
tera-	10^{12}	T	tebi	2^{40}	Ti
peta-	10^{15}	P	pebi	2^{50}	Pi
exa-	10^{18}	E	exbi	2^{60}	Ei
zetta-	10^{21}	Z	zebi	2^{70}	Zi
yotta-	10^{24}	Y	yobi	2^{80}	Yi

Amendment 2 to “IEC 60027-2: Letter symbols to be used in electrical technology – Part 2: Telecommunications and electronics” (1999)

Výsledky měření

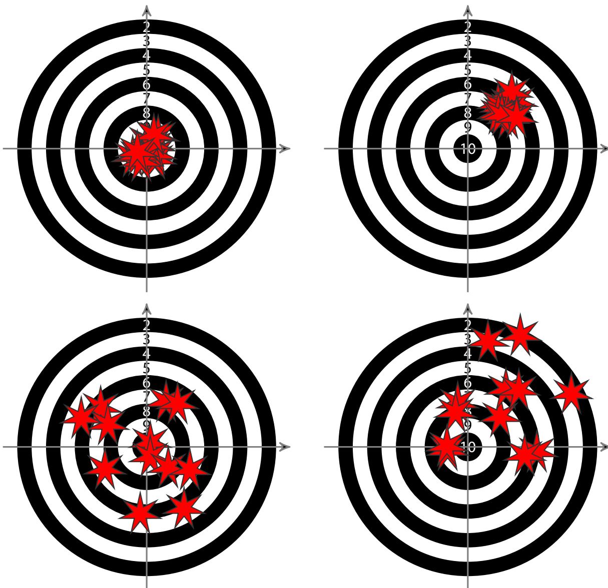
$$x = (\hat{\mu}_x \pm z_x) \text{ [jednotka]}$$

- $\hat{\mu}_x$… nejpravděpodobnější hodnota měřené veličiny
- z_x… interval spolehlivosti / přesnost
- jak tyto věci spočítat / odhadnout?

Chyby měření

- Klasifikace chyb podle místa vzniku
  - instrumentální (přístrojové) chyby
  - metodické chyby
  - teoretické chyby (principy, model)
  - chyby zpracování
- Klasifikace chyb podle původu
  - hrubé (omyly)
  - systematické
  - náhodné

Chyby měření

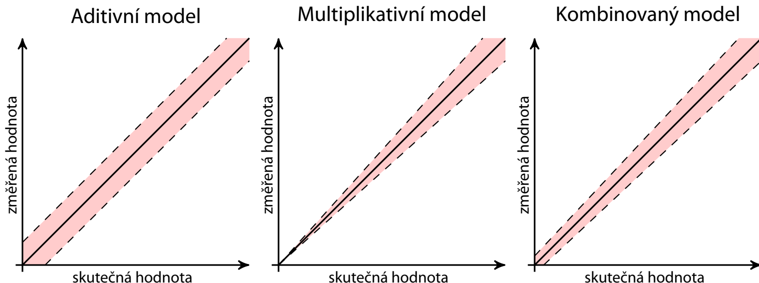


Přesnost měřících nástrojů

Přesnost přístroje … náhodná chyba

Správnost přístroje … systematická chyba

- Aditivní vs. multiplikativní chyby
- Mezní hodnota chyb
- Třída přesnosti přístroje



Hrubé chyby

- Hrubé chyby se musí ze sady měření vyloučit
- Volba měřící metody / měřících metod – příklad pro Javu
  - Problém garbage collection
    - `-verbose:gc`
    - krátká měření: vybrat pouze běhy, v nichž nedošlo ke GC
    - dlouhé běhy: dostatečně dlouhé, aby se přítomnost GC projevila representativně
  - Problém HotSpot kompilace
    - `-XX:+PrintCompilation`
    - dostatečný warm-up (minuty!)
    - mohou se vyskytovat rekompilace (optimalizace, nahrání nové třídy která zruší dosavadní předpoklady)
    - housekeeping tasks: oddělení nesouvisejících měření pauzou nebo restartem JVM

Náhodné chyby

aneb proč se běžně pracuje s normálním rozdělením chyb?

- Hypotéza elementárních chyb (Horák, 1958)
  - každá náhodná chyba v měření je složena z řady malých chyb
  - při velkém počtu měření se vyskytne zhruba stejný počet chyb kladných i záporných a malé chyby jsou početnější než velké

1. m elementárních náhodných vlivů
2. každý elementární vliv generuje chybu α (dále označováno jako případ *a*) nebo $-\alpha$ (dále případ *b*)
3. chyby *a* a *b* jsou stejně časté

- dostáváme binomické rozdělení kumulace vlivů elementárních chyb
- pro velké množství náhodných jevů se blíží rozdělení normálnímu

Náhodné chyby

aneb proč se běžně pracuje s normálním rozdělením chyb?

- Co se stane, pokud $m \to \infty$?
  - binomické rozdělení

$$\binom{m}{0}a^m, \binom{m}{1}a^{m-1}b, \ldots, \binom{m}{l}a^{m-l}b^l, \ldots, \binom{m}{m}b^m$$

$$P(0) = \frac{1}{2^m}\binom{m}{m/2} \qquad P(\varepsilon_l) = \frac{1}{2^m}\binom{m}{l}, \varepsilon_l = (l-(m-l))\alpha = (2l-m)\alpha = 2s\alpha$$

  - pro sudá $m = 2k \implies k \to \infty$ (sudá, abychom měli $P(0)$)

$$P(\varepsilon) = P(2s\alpha) = \frac{1}{2^{2k}}\binom{2k}{k+s}$$

$$\frac{P(2s\alpha)}{P(0)} = \frac{\binom{2k}{k+s}}{\binom{2k}{k}} = \frac{k(k-1)\cdots(k-s+1)}{(k+1)(k+2)\cdots(k+s)} = \frac{\left(1-\frac{1}{k}\right)\left(1-\frac{2}{k}\right)\cdots\left(1-\frac{s-1}{k}\right)}{\left(1+\frac{1}{k}\right)\left(1+\frac{2}{k}\right)\cdots\left(1+\frac{s}{k}\right)}$$

  - pro $s \ll k$

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots \approx x$$

$$\ln\frac{P(2s\alpha)}{P(0)} = -\frac{1}{k} - \frac{2}{k} - \cdots - \frac{s-1}{k} - \frac{1}{k} - \frac{2}{k} - \cdots - \frac{s}{k} = -\frac{2}{k}\frac{s(s-1)}{2} - \frac{s}{k} = -\frac{s^2}{k}$$

$$P(2s\alpha) = P(0)\mathrm{e}^{-\frac{s^2}{k}} = P(0)\mathrm{e}^{-\frac{\varepsilon^2}{4k\alpha^2}}$$

Studentovo rozdělení t

- Používá se pro normální rozdělení při malém vzorku

$$f(t) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\,\Gamma\left(\frac{\nu}{2}\right)} \left(1 + \frac{t^2}{\nu}\right)^{-(\nu+1)/2}$$

kde ν je počet stupňů volnosti.

- odhad průměrů a chyby
- t-test – odlišení průměrů

Odhad spolehlivosti

$$x = (\hat{\mu}_x \pm z_x) \text{ [jednotka]}$$

Statistická definice (Šťastný, 1997): Je-li výsledek měření $\hat{\mu}_x$ a z_x je chyba tohoto měření odpovídající míře jistoty p, pak skutečná hodnota měřené veličiny leží v intervalu $(\hat{\mu}_x \pm z_x)$ s pravděpodobností p.

- Intervaly
  - 0,68 – střední kvadratická chyba
  - 0,95
  - 0,99 – krajní chyba
- Zaokrouhlování
  - z_x nejvýše na 2 platná místa
  - $\hat{\mu}_x$ podle z_x

Odhad spolehlivosti

$$x = (\hat{\mu}_x \pm z_x)\ [\text{jednotka}]$$

Pro normální rozdělení chyby

- $\hat{\mu}_x = \overline{x} = \frac{\sum_{i=1}^{N} x_i}{n}$
- s směrodatná odchylka jednoho měření, D rozptyl

$$s = \sqrt{D} = \sqrt{\frac{\sum_{i=1}^{N}(\overline{x} - x_i)^2}{n-1}}$$

- $s_{\overline{x}} = \sqrt{\sum_{i=1}^{N}(\frac{1}{n})^2 s_{x_i}}$ a protože měření byly prováděny za stejných podmínek

$$s_{\overline{x}} = \frac{s_x}{\sqrt{n}} = \sqrt{\frac{\sum_{i=1}^{N}(\overline{x} - x_i)^2}{n(n-1)}}$$

Odhad spolehlivosti

$$x = (\hat{\mu}_x \pm z_x)\ [\text{jednotka}]$$

Pro normální rozdělení chyby

- $z_x = t_{(p;n-1)} s_{\overline{x}}$

n \ P	0,683	0,954	0,99	n \ P	0,683	0,954	0,99
1	1,8395	13,8155	63,6567	16	1,0329	2,1633	2,9208
2	1,3224	4,5001	9,9248	18	1,0292	2,1433	2,8784
3	1,1978	3,2923	5,8409	20	1,0263	2,1276	2,8453
4	1,1425	2,8585	4,6041	30	1,0176	2,0817	2,75
5	1,1113	2,6396	4,0321	40	1,0133	2,0595	2,7045
6	1,0913	2,5084	3,7074	50	1,0108	2,0463	2,6778
7	1,0775	2,4214	3,4995	60	1,0091	2,0377	2,6603
8	1,0673	2,3594	3,3554	70	1,0078	2,0315	2,6479
9	1,0594	2,3131	3,2498	80	1,0069	2,0269	2,6387
10	1,0533	2,2773	3,1693	90	1,0062	2,0234	2,6316
12	1,0441	2,2253	3,0545	100	1,0057	2,0206	2,6259
14	1,0377	2,1895	2,9768				

Odhad spolehlivosti

$$x = (\hat{\mu}_x \pm z_x) \text{ [jednotka]}$$

Příklad – měření výšky válečku (Šťastný, 1997):

výška v [mm]	4,6	4,5	4,7	4,4	4,5	4,6	4,4	4,4	4,3	4,5

- $n = 10$
- $\overline{v} = 4{,}49$ [mm]
- $s_{\overline{v}} = 0{,}038$ [mm]
- $t_{(0{,}68;9)} = 1{,}059$
- $t_{(0{,}99;9)} = 3{,}250$

$$v = (4{,}49 \pm 0{,}04) \text{ mm} \quad \text{pro } p = 0{,}68$$
$$v = (4{,}49 \pm 0{,}12) \text{ mm} \quad \text{pro } p = 0{,}99$$

Zákon přenosu chyb

- Na základě Taylorova rozvoje do druhého členu

$$s_{\overline{z}}^2 = \sum_{i=1}^{N} \left(\frac{\partial z}{\partial x_i} \right)^2 s_{\overline{x_i}}^2 + 2 \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{\partial z}{\partial x_i} \frac{\partial z}{\partial x_j} s_{\overline{x_i}} s_{\overline{x_j}} \varrho_{ij} \, ,$$

kde $s_{\overline{x_i}}^2$ je rozptyl (variance) x_i a ϱ_{ij} je kovariance x_i a x_j.

Pro jednoduché případy, kdy x a y jsou nezávislé ($\varrho_{ij} = 0$):

- aditivní funkce $z = ax \pm by$

$$s_{\overline{z}} = \sqrt{a^2 s_{\overline{x}}^2 + b^2 s_{\overline{y}}^2} \, , \tag{1}$$

- multiplikativní funkce $z = ax^b y^c$

$$s_{\overline{z}} = \overline{z} \sqrt{\left(\frac{b s_{\overline{x}}}{\overline{x}} \right)^2 + \left(\frac{c s_{\overline{y}}}{\overline{y}} \right)^2} \, . \tag{2}$$

kde $\overline{z} = a\overline{x}^b \overline{y}^c$, protože

$$\sum_{i=1}^{N} \left(\frac{\partial z}{\partial x_i} \right)^2 s_{\overline{i}}^2 = \left(\frac{abx^b y^c s_{\overline{x}}}{x} \right)^2 + \left(\frac{ax^b c y^c s_{\overline{y}}}{y} \right)^2 = z^2 \left(\left(\frac{b s_{\overline{x}}}{x} \right)^2 + \left(\frac{c s_{\overline{y}}}{y} \right)^2 \right)$$

- Příklad použití: **`http://www.phy.ohiou.edu/~murphy/courses/sample.pdf`**

Skripta Fr. Šťastného (Šťastný, 1997)
`http://amper.ped.muni.cz/jenik/nejistoty/`