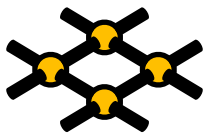
Vláknové programování
část IX

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



Laboratoř pokročilých síťových technologií

PV192
2010–03–23

Přehled přednášky

ThreadPoolExecutors Revisited

Uváznutí

Optimalizace výkonu

Paměťový model Javy

(příklady povětšinou převzaty z JCiP, Goetz)

ThreadFactory

- TPE vytváří vlákna pomocí ThreadFactory
  - default ThreadFactory: nedémonická, bez speciálních nastavení
- Možnost předefinovat, jak se budou vytvářet vlákna
  - nastavení pojmenování vláken
  - vlastní třída vytvářených vláken (statistiky, ladění)
  - specifikace vlastního `UncaughtExceptionHandler`
  - nastavení priorit (raději nedělat)
  - nastavení démonického stavu (raději nedělat)
  - v případě použití bezpečnostních politik (security policies) lze použít `privilegedThreadFactory`
    - podědění oprávnění, `AccessControlContext` a `contextClassLoader` od vlákna vytvářejícího `privilegedThreadFactory`

ThreadFactory

```
public class MyThreadFactory implements ThreadFactory {
    private final String poolName;
    class MyAppThread extends Thread {
        public MyAppThread(Runnable runnable, String poolName) {
            super(runnable, poolName);
        }
    }

    public MyThreadFactory(String poolName) {
        this.poolName = poolName;
    }

    public Thread newThread(Runnable runnable) {
        return new MyAppThread(runnable, poolName);
    }
}
```

Modifikace Executorů za běhu

- settery a gettery na různé vlastnosti
- možnost přetypování executorů vyrobených přes factory metody (kromě `newSingleThreadExecutor`) na `ThreadPoolExecutor`
- omezení modifikací
  - nechceme nechat vývojáře šťourat do svých TPE
  - factory metoda `Executor.unconfigurableExecutorService`
    - bere `ExecutorService`
    - vrací omezenou `ExecutorService` pomocí `DelegatedExecutorService`, která rozšiřuje `AbstractExecutorService`
  - využívání metodou `newSingleThreadExecutor`

Modifikace TPE

- Háčky pro modifikace
  - `beforeExecute`
  - `afterExecute`
  - `terminate`
- Např. sběr statistik

Modifikace TPE

```
public class TimingThreadPool extends ThreadPoolExecutor {

    public TimingThreadPool() {
        super(1, 1, 0L, TimeUnit.SECONDS, null);
    }

    private final ThreadLocal<Long> startTime = new ThreadLocal<Long>();
    private final Logger log = Logger.getLogger("TimingThreadPool");
    private final AtomicLong numTasks = new AtomicLong();
    private final AtomicLong totalTime = new AtomicLong();

    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
        log.fine(String.format("Thread %s: start %s", t, r));
        startTime.set(System.nanoTime());
    }
```

Modifikace TPE

```
    protected void afterExecute(Runnable r, Throwable t) {
        try {
            long endTime = System.nanoTime();
            long taskTime = endTime - startTime.get();
            numTasks.incrementAndGet();
            totalTime.addAndGet(taskTime);
            log.fine(String.format("Thread %s: end %s, time=%dns",
                    t, r, taskTime));
        } finally {
            super.afterExecute(r, t);
        }
    }

    protected void terminated() {
        try {
            log.info(String.format("Terminated: avg time=%dns",
                    totalTime.get() / numTasks.get()));
        } finally {
            super.terminated();
        }
    }
}
```

Možné řešení kvízu

```
public class DynamicTPE {
    static class CustomTPE extends ThreadPoolExecutor {
        final int userProvidedPoolSize;
        final int queueCapacity;

        CustomTPE(int corePoolSize, int maximumPoolSize,
                  long keepAliveTime, TimeUnit unit,
                  BlockingQueue<Runnable> workQueue) {
            super(corePoolSize, maximumPoolSize,
                    keepAliveTime, unit, workQueue);
            userProvidedPoolSize = corePoolSize;
            queueCapacity = workQueue.size()
                    + workQueue.remainingCapacity();
        }

        @Override public Future<?> submit(Runnable task) {
            autoAdjustCorePoolSize();
            return super.submit(task);
        }

        @Override synchronized public void setCorePoolSize
                (int corePoolSize) {
            super.setCorePoolSize(corePoolSize);
        }

        @Override synchronized public void setMaximumPoolSize
                (int maximumPoolSize) {
            super.setMaximumPoolSize(maximumPoolSize);
        }
```

Možné řešení kvízu

```
1       synchronized private void autoAdjustCorePoolSize() {
            final int queueRemaining = getQueue().remainingCapacity();
3           final int extension;
            if (queueRemaining < 0.25 * queueCapacity) {
5               extension = (int) Math.round(0.25 * queueCapacity
                        - queueRemaining);
7               if (getCorePoolSize() + extension < getMaximumPoolSize())
                    setCorePoolSize(getCorePoolSize() + extension);
9               else
                    setCorePoolSize(getMaximumPoolSize());
11          }
            else if (queueRemaining > 0.75 * queueCapacity) {
13              extension = (int) Math.round(queueRemaining
                        - 0.75 * queueCapacity);
15              if (getCorePoolSize() - extension > userProvidedPoolSize)
                    setCorePoolSize(getCorePoolSize() - extension);
17              else
                    setCorePoolSize(userProvidedPoolSize);
19          }
        }
21
    }
```

Možné řešení kvízu

```
    public static void main(String[] args) {
        ThreadPoolExecutor tpe = new CustomTPE(1, 10, 60, TimeUnit.SECONDS, new
        tpe.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy(
        ArrayList<Future> taskList = new ArrayList<Future>();
        Runnable r = new Runnable() {
            public void run() {
                try {
                    Thread.sleep(30);
                } catch (InterruptedException e) {
                }
                System.out.println("bla");
            }
        };
        for (int i = 0; i < 1000; i++) {
            Future f = tpe.submit(r);
            taskList.add(f);
            System.out.println("TPE corepoolsize: " + tpe.getCorePoolSize());
            System.out.println("TPE poolsize: " + tpe.getPoolSize());
        }
        for (Future future : taskList) {
            try {
                Object o = future.get();
            } catch (InterruptedException e) {
            } catch (ExecutionException e) {
            }
        }
        tpe.shutdown();
    }
```

Možné řešení kvízu

```
1  TPE corepoolsize: 1
   TPE poolsize: 1
3  TPE corepoolsize: 1
   TPE poolsize: 1
5  TPE corepoolsize: 1
   TPE poolsize: 1
7  TPE corepoolsize: 1
   TPE poolsize: 1
9  TPE corepoolsize: 1
   TPE poolsize: 1
11 TPE corepoolsize: 1
   TPE poolsize: 1
13 TPE corepoolsize: 1
   TPE poolsize: 1
15 TPE corepoolsize: 1
   TPE poolsize: 1
17 TPE corepoolsize: 1
   TPE poolsize: 1
19 TPE corepoolsize: 1
   TPE poolsize: 1
21 TPE corepoolsize: 1
   TPE poolsize: 1
23 TPE corepoolsize: 1
```

```
   TPE poolsize: 1
2  TPE corepoolsize: 1
   TPE poolsize: 1
4  TPE corepoolsize: 1
   TPE poolsize: 1
6  TPE corepoolsize: 1
   TPE poolsize: 1
8  TPE corepoolsize: 1
   TPE poolsize: 1
10 TPE corepoolsize: 1
   TPE poolsize: 1
12 TPE corepoolsize: 2
   TPE poolsize: 2
14 TPE corepoolsize: 4
   TPE poolsize: 4
16 TPE corepoolsize: 7
   TPE poolsize: 7
18 TPE corepoolsize: 10
   TPE poolsize: 10
20 TPE corepoolsize: 10
   TPE poolsize: 10
22 bla
   bla
```

Možné řešení kvízu

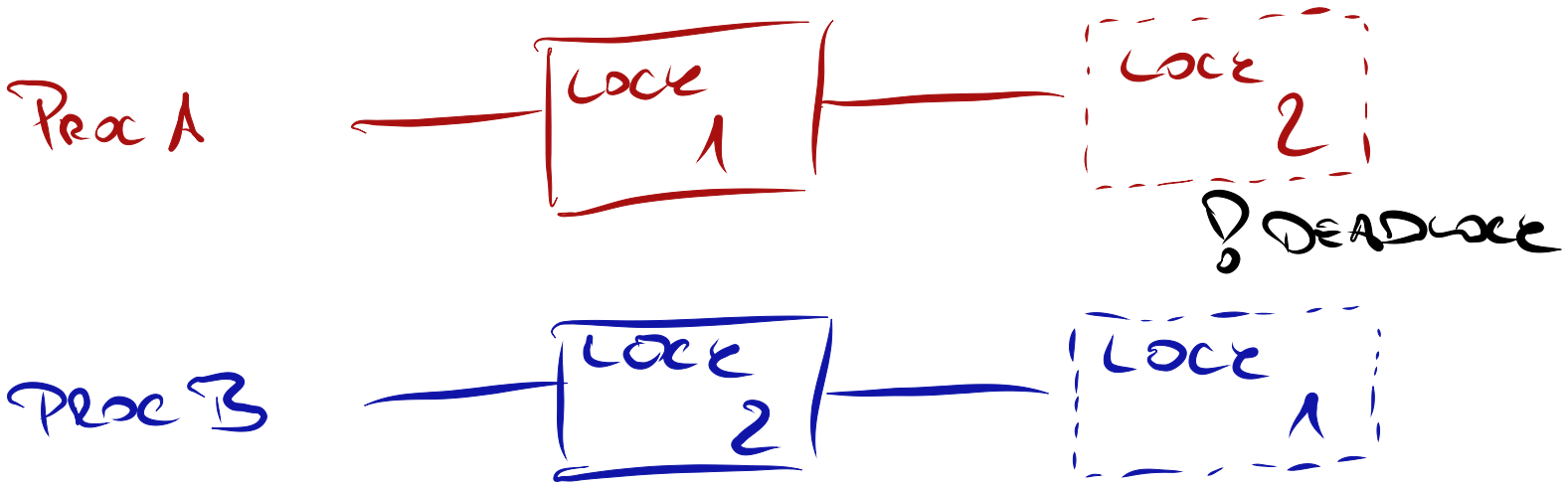
- Stojí nám to za to?
  - drahá synchronizace
  - režije se startováním a ukončováním vláken
  - nestačí vhodně nastavit `corePoolSize` a `allowCoreThreadTimeOut`?

Kompletně vlastní implementace TPE

- Zdrojové kódy:
  - **`http://kickjava.com/src/java/util/concurrent/ThreadPoolExecutor.java.htm`**
  - **`http://kickjava.com/src/java/util/concurrent/ScheduledThreadPoolExecutor.java.htm`**

Deadlock

- Deadlock – uváznutí, smrtelné objetí ;-)
- Vzájemné nekončící čekání na zámky



- Potřeba globálního uspořádání zámků
  - zamykání podle globálního uspořádání
- Možnost využití `Lock.tryLock()`
  - náhodný rovnoměrný back-off
  - náhodný exponenciální back-off
  - nelze použít s monitory
- Řešení deadlocků runtimem (ne v Javě)

Deadlock

```
    public static void transferMoney(Account fromAccount,
2                                    Account toAccount,
                                     DollarAmount amount)
4           throws InsufficientFundsException {
        synchronized (fromAccount) {
6           synchronized (toAccount) {
                if (fromAccount.getBalance().compareTo(amount) < 0)
8                   throw new InsufficientFundsException();
                else {
10                  fromAccount.debit(amount);
                    toAccount.credit(amount);
12              }
            }
14      }
    }
```

Deadlock

```
    public static void transferMoney(Account fromAccount,
2                                    Account toAccount,
                                     DollarAmount amount)
4           throws InsufficientFundsException {
        synchronized (fromAccount) {
6           synchronized (toAccount) {
                if (fromAccount.getBalance().compareTo(amount) < 0)
8                   throw new InsufficientFundsException();
                else {
10                  fromAccount.debit(amount);
                    toAccount.credit(amount);
12              }
            }
14      }
    }
```

Deadlock

```
    public void transferMoney(final Account fromAcct,
2                             final Account toAcct,
                              final DollarAmount amount)
4           throws InsufficientFundsException {
        class Helper {
6           public void transfer() throws InsufficientFundsException {
                if (fromAcct.getBalance().compareTo(amount) < 0)
8                   throw new InsufficientFundsException();
                else {
10                  fromAcct.debit(amount);
                    toAcct.credit(amount);
12              }
            }
14      }
        int fromHash = System.identityHashCode(fromAcct);
16      int toHash = System.identityHashCode(toAcct);
```

- `System.identityHashCode(o)` může vrátit pro dva různé objekty identický hash
  - řídký problém

Deadlock

```
        if (fromHash < toHash) {
2           synchronized (fromAcct) {
                synchronized (toAcct) {
4                   new Helper().transfer();
                }
6           }
        } else if (fromHash > toHash) {
8           synchronized (toAcct) {
                synchronized (fromAcct) {
10                  new Helper().transfer();
                }
12          }
        } else {
14          synchronized (tieLock) {
                synchronized (fromAcct) {
16                  synchronized (toAcct) {
                        new Helper().transfer();
18                  }
                }
20          }
        }
```

Deadlock

```
    private static Random rnd = new Random();
2
    public boolean transferMoney(Account fromAcct,
4                                Account toAcct,
                                 DollarAmount amount,
6                                long timeout,
                                 TimeUnit unit)
8           throws InsufficientFundsException, InterruptedException {
        long fixedDelay = getFixedDelayComponentNanos(timeout, unit);
10      long randMod = getRandomDelayModulusNanos(timeout, unit);
        long stopTime = System.nanoTime() + unit.toNanos(timeout);
```

Deadlock

```
        while (true) {
            if (fromAcct.lock.tryLock()) {
                try {
                    if (toAcct.lock.tryLock()) {
                        try {
                            if (fromAcct.getBalance().compareTo(amount) < 0)
                                throw new InsufficientFundsException();
                            else {
                                fromAcct.debit(amount);
                                toAcct.credit(amount);
                                return true;
                            }
                        } finally {
                            toAcct.lock.unlock();
                        }
                    }
                } finally {
                    fromAcct.lock.unlock();
                }
            }
            if (System.nanoTime() < stopTime)
                return false;
            NANOSECONDS.sleep(fixedDelay + rnd.nextLong() % randMod);
        }
```

Otevřená volání

```
1   class Taxi {
        @GuardedBy("this") private Point location, destination;
3       private final Dispatcher dispatcher;

5       public Taxi(Dispatcher dispatcher) {
            this.dispatcher = dispatcher;
7       }

9       public synchronized Point getLocation() {
            return location;
11      }

13      public synchronized void setLocation(Point location) {
            this.location = location;
15          if (location.equals(destination))
                dispatcher.notifyAvailable(this);
17      }

19      public synchronized Point getDestination() {
            return destination;
21      }

23      public synchronized void setDestination(Point destination) {
            this.destination = destination;
25      }
    }
```

Otevřená volání

```
class Dispatcher {
    @GuardedBy("this") private final Set<Taxi> taxis;
    @GuardedBy("this") private final Set<Taxi> availableTaxis;

    public Dispatcher() {
        taxis = new HashSet<Taxi>();
        availableTaxis = new HashSet<Taxi>();
    }

    public synchronized void notifyAvailable(Taxi taxi) {
        availableTaxis.add(taxi);
    }

    public synchronized Image getImage() {
        Image image = new Image();
        for (Taxi t : taxis)
            image.drawMarker(t.getLocation());
        return image;
    }
}
```

Otevřená volání

- Otevřené volání (open call)
  - volání cizí, které není obaleno žádnou synchronizací
  - preferovaný způsob
- Převod na otevřené volání
  - synchronizace by měla být omezena na lokální proměnné
  - problém se zachováním sémantiky
- Možnost globálního zámku

Otevřená volání

```
class Taxi {
    @GuardedBy("this") private Point location, destination;
    private final Dispatcher dispatcher;

    public Taxi(Dispatcher dispatcher) { this.dispatcher = dispatcher; }

    public synchronized Point getLocation() { return location; }

    public void setLocation(Point location) {
        boolean reachedDestination;
        synchronized (this) {
            this.location = location;
            reachedDestination = location.equals(destination);
        }
        if (reachedDestination)
            dispatcher.notifyAvailable(this);
    }

    public synchronized Point getDestination() { return destination; }

    public synchronized void setDestination(Point destination) {
        this.destination = destination;
    }
}
```

Otevřená volání

```
    class Dispatcher {
2       @GuardedBy("this") private final Set<Taxi> taxis;
        @GuardedBy("this") private final Set<Taxi> availableTaxis;
4
        public Dispatcher() {
6           taxis = new HashSet<Taxi>();
            availableTaxis = new HashSet<Taxi>();
8       }

10      public synchronized void notifyAvailable(Taxi taxi) {
            availableTaxis.add(taxi);
12      }

14      public Image getImage() {
            Set<Taxi> copy;
16          synchronized (this) {
                copy = new HashSet<Taxi>(taxis);
18          }
            Image image = new Image();
20          for (Taxi t : copy)
                image.drawMarker(t.getLocation());
22          return image;
        }
24  }
```

Hladovění

- Hladovění (starvation) nastává, pokud je vláknu neustále odpírán zdroj, který je potřeba k dalšímu postupu
  - běžné použití zámků je férové
  - problém při nastavování priorit

```
2         t.setPriority(Thread.MIN_PRIORITY); // 1
          t.setPriority(Thread.NORM_PRIORITY); // 5
          t.setPriority(Thread.MAX_PRIORITY); // 10
```

    - problém platformové závislosti priorit
    - možná pomoc pro zvýšení responsivity GUI
  - typické pokusy o „řešení“ problémů

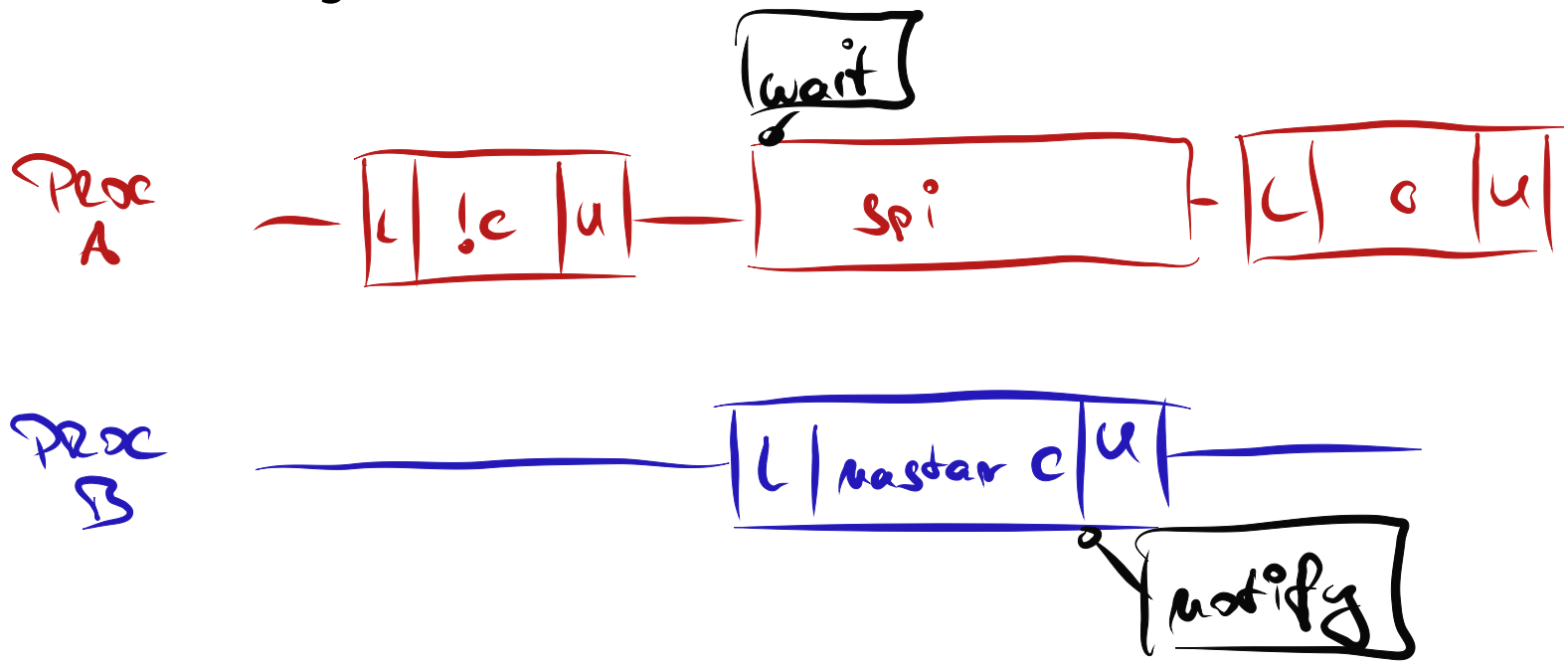
```
1         Thread.yield();
          Thread.sleep(100);
```

Další typy uváznutí

- Livelock
  - uváznutí, při němž se vlákno (aktivně) snaží o činnosti, která opakovaně selhává
  - náhodnostní exponenciální back-off
- Ztracené právy
  - `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

Další typy uváznutí

- Podmíněná signalizace



Další typy uváznutí

```
        // 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

Hledání problémů

- Výpis stavu JVM
  - **`SIGQUIT`** na unixech (ev. Ctrl-\ pokud mapuje na **`SIGQUIT`**)
  - Ctrl-Break na Windows

Hledání problémů

```
    public static void main(String[] args) {
2       final Object a = new Object();
        final Object b = new Object();
4
        Thread t1 = new Thread(new Runnable() {
6           public void run() {
                try {
8                   synchronized (a) {
                        Thread.sleep(1000);
10                      System.out.println("t1 - cekam na b");
                        synchronized (b) {
12                          System.out.println("t1 - jsem zde");
                        }
14                  }
                } catch (InterruptedException e) {
16              }
            }
18      });
```

Hledání problémů

```
        Thread t2 = new Thread(new Runnable() {
2           public void run() {
                try {
4                   synchronized (b) {
                        Thread.sleep(1000);
6                       System.out.println("t2 - cekam na a");
                        synchronized (a) {
8                           System.out.println("t2 - jsem zde");
                        }
10                  }
                } catch (InterruptedException e) {
12              }
            }
14      });

16      t1.start();
        t2.start();
```

Hledání problémů

```
$ java IntentionalDeadlock
t2 - cekam na a
t1 - cekam na b
2010-04-22 11:46:25
Full thread dump Java HotSpot(TM) Client VM (16.2-b04 mixed mode, sharing):

"DestroyJavaVM" prio=6 tid=0x020b1000 nid=0x164c waiting on condition [0x00000000]
   java.lang.Thread.State: RUNNABLE

"Thread-1" prio=6 tid=0x02149800 nid=0x1b4c waiting for monitor entry [0x0480f000]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at IntentionalDeadlock$2.run(IntentionalDeadlock.java:35)
        - waiting to lock <0x243e6928> (a java.lang.Object)
        - locked <0x243e6930> (a java.lang.Object)
        at java.lang.Thread.run(Unknown Source)

"Thread-0" prio=6 tid=0x02146c00 nid=0x1a38 waiting for monitor entry [0x0477f000]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at IntentionalDeadlock$1.run(IntentionalDeadlock.java:20)
        - waiting to lock <0x243e6930> (a java.lang.Object)
        - locked <0x243e6928> (a java.lang.Object)
        at java.lang.Thread.run(Unknown Source)
```

Hledání problémů

```
"Low Memory Detector" daemon prio=6 tid=0x02121400 nid=0xbd8 runnable [0x00000000]
   java.lang.Thread.State: RUNNABLE

"CompilerThread0" daemon prio=10 tid=0x02119800 nid=0x1708 waiting on condition [0x00000000]
   java.lang.Thread.State: RUNNABLE

"Attach Listener" daemon prio=10 tid=0x02118400 nid=0x13d0 runnable [0x00000000]
   java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" daemon prio=10 tid=0x02115400 nid=0x5a0 waiting on condition [0x00000000]
   java.lang.Thread.State: RUNNABLE
```

```
Heap
 def new generation   total 4928K, used 466K [0x243b0000, 0x24900000, 0x29900000)
  eden space 4416K,  10% used [0x243b0000, 0x24424828, 0x24800000)
  from space 512K,   0% used [0x24800000, 0x24800000, 0x24880000)
  to   space 512K,   0% used [0x24880000, 0x24880000, 0x24900000)
 tenured generation   total 10944K, used 0K [0x29900000, 0x2a3b0000, 0x343b0000)
  the space 10944K,   0% used [0x29900000, 0x29900000, 0x29900200, 0x2a3b0000)
 compacting perm gen  total 12288K, used 42K [0x343b0000, 0x34fb0000, 0x383b0000)
  the space 12288K,   0% used [0x343b0000, 0x343ba960, 0x343baa00, 0x34fb0000)
    ro space 10240K,  51% used [0x383b0000, 0x388dae00, 0x388dae00, 0x38db0000)
    rw space 12288K,  54% used [0x38db0000, 0x394472d8, 0x39447400, 0x399b0000)
```

Hledání problémů

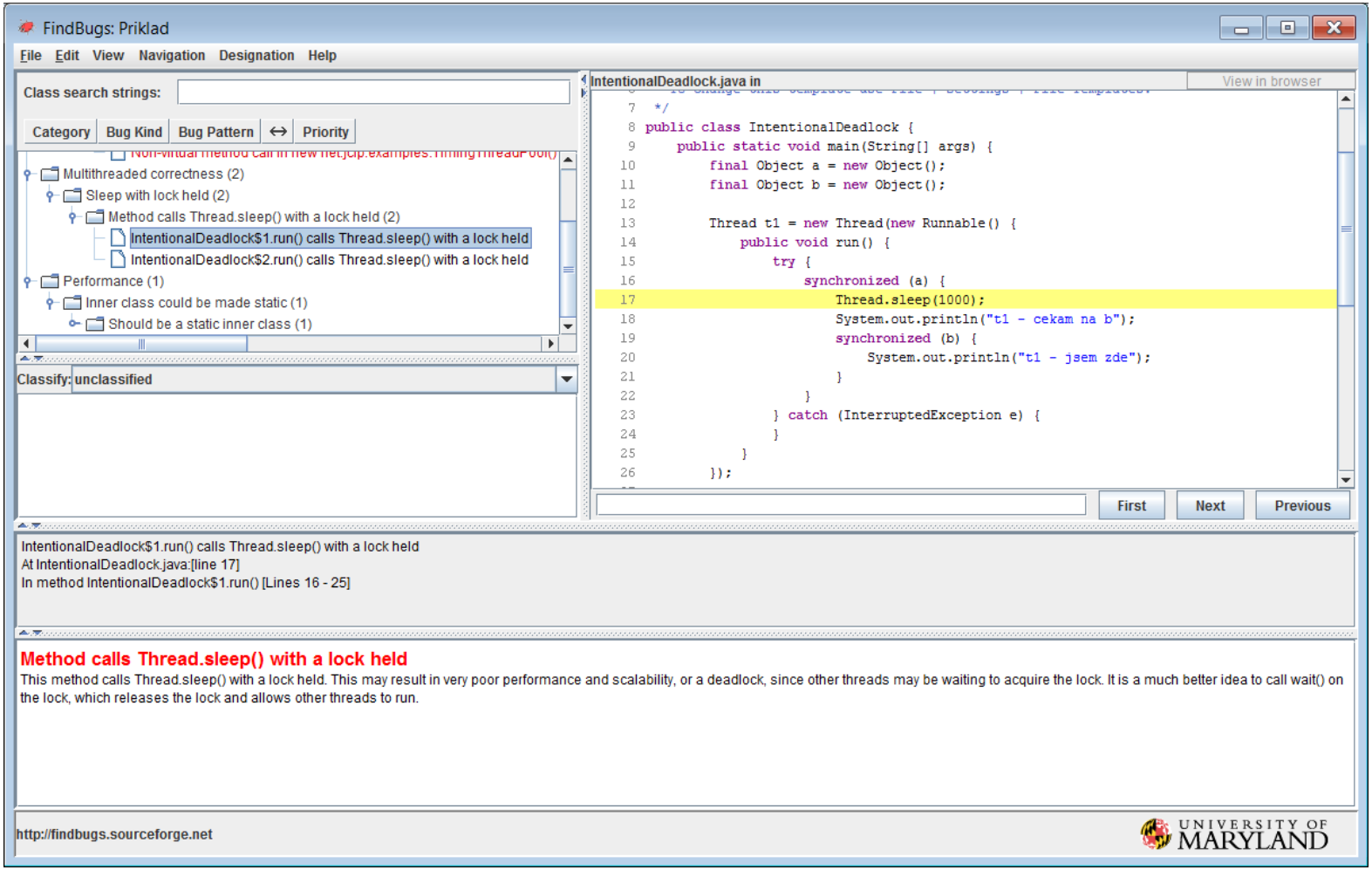
```
Found one Java-level deadlock:
============================
"Thread-1":
  waiting to lock monitor 0x020d53ac (object 0x243e6928, a java.lang.Object),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x020d6c74 (object 0x243e6930, a java.lang.Object),
  which is held by "Thread-1"

Java stack information for the threads listed above:
===================================================
"Thread-1":
        at IntentionalDeadlock$2.run(IntentionalDeadlock.java:35)
        - waiting to lock <0x243e6928> (a java.lang.Object)
        - locked <0x243e6930> (a java.lang.Object)
        at java.lang.Thread.run(Unknown Source)
"Thread-0":
        at IntentionalDeadlock$1.run(IntentionalDeadlock.java:20)
        - waiting to lock <0x243e6930> (a java.lang.Object)
        - locked <0x243e6928> (a java.lang.Object)
        at java.lang.Thread.run(Unknown Source)

Found 1 deadlock.
```

Statická analýza kódu

- FindBugs
  `http://findbugs.sourceforge.net/`



Anotace

- Vícečlenný tým programátorů – předávání myšlenek
  - komentáře v kódy
  - anotace
    - anotace se dají použít i pro statickou analýzu kódu

```
import net.jcip.annotations.GuardedBy;
// http://www.javaconcurrencyinpractice.com/jcip-annotations.jar
```

Anotace

- Anotace tříd
  - `@Immutable`
  - `@ThreadSafe`
  - `@NotThreadSafe`
- Anotace polí
  - `@GuardedBy("this")`
    - monitor (intrinsic lock) na this
  - `@GuardedBy("jmenoPole")`
    - explicitní zámek na `jmenoPole` pokud je potomkem `Lock`
    - jinak monitor na `jmenoPole`
  - `@GuardedBy("JmenoTridy.jmenoPole")`
    - obdobné, odkazuje se statické pole jiné třídy
  - `@GuardedBy("jmenoMetody()")`
    - metoda `jmenoMetody()` vrací zámek
  - `@GuardedBy("JmenoTridy.class")`
    - literál třídy (objekt) pro pojmenovanou třídu

Omezování zámků

$$\text{zrychlení} \leq \frac{1}{s + \frac{1-s}{n}}$$

- JVM se snaží dělat
  - eliminaci synchronizací, které nemohou nastat (např. pomocí escape analysis)
  - kombinace více zámků do jednoho (lock coarsening)
- Zbytečně nesynchronizovat
  - delegace bezpečnosti (thread safety delegation)
  - omezení rozsahu synchronizace (get in – get out principle, např. Taxi/Dispatcher)
  - dělení zámků (lock splitting) – pouze pro **nezávislé** proměnné/objekty
  - ořezávání zámků (lock stripping)
  - RW zámky
- Neprovádět object pooling na jednoduchých objektech
  - `new` je levnější jako `malloc`

```
2       synchronized (new Object()) {
            System.out.println("bleeee");
        }
```

Omezování zámků

$$\text{zrychlení} \leq \frac{1}{s + \frac{1-s}{n}}$$

- JVM se snaží dělat
  - eliminaci synchronizací, které nemohou nastat (např. pomocí escape analysis)
  - kombinace více zámků do jednoho (lock coarsening)
- Zbytečně nesynchronizovat
  - delegace bezpečnosti (thread safety delegation)
  - omezení rozsahu synchronizace (get in – get out principle, např. Taxi/Dispatcher)
  - dělení zámků (lock splitting) – pouze pro **nezávislé** proměnné/objekty
  - ořezávání zámků (lock stripping)
  - RW zámky
- Neprovádět object pooling na jednoduchých objektech
  - `new` je levnější jako `malloc`

```
2       synchronized (new Object()) {
            System.out.println("bleeee");
        }
```

Omezování zámků

- Podobně jako dělení zámků, ale pro proměnný počet nezávislých proměnných/objektů
- Příklad ořezávání zámků – `ConcurrentHashMap`
  - 16 zámků
  - každý z N hash buckets je chráněný zámkem $N \mod 16$
  - předpokládáme rovnoměrné rozdělení položek mezi kbelíky

  $\Longrightarrow$ 16 paralelních přístupů

  $\Longrightarrow$ přístup k celé kolekci vyžaduje všech 16 zámků
  - rozdělení kumulativních polí do jednotlivých kbelíků

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

Několik rad a myšlenek k měření

- *Einmal is keinmal!*
- Průměr

$$\overline{x} = \frac{\sum_{i=1}^{N} x_i}{N} \tag{1}$$

- Směrodatná neboli standardní odchylka

$$\sigma_x = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2} = \sqrt{\frac{1}{N-1}\left(\sum_{i=1}^{N} x_i^2 - N\overline{x}^2\right)} \tag{2}$$

- Směrodatná odchylka aritmetického průměru

$$\sigma_{\overline{x}} = \frac{\sigma_x}{\sqrt{N}} \tag{3}$$

Ze směrodatné odchylky aritmetického průměru mimo jiné také plyne její závislost na počtu vzorků, tj. chceme-li zvýšit přesnost dvakrát, musíme zvětšit velikost vzorku čtyřikrát.

Několik rad a myšlenek k měření

- Odhad spolehlivosti pro normální rozdělení

Pointerval	Úroveň spolehlivosti [%]
σ	68.27
1.96σ	95.0
2σ	95.45
3σ	99.73
4σ	99.99

Běžně se uvádí přesnost na 95 %, tedy pro normální rozdělení

$$\overline{x} \pm 1.96\sigma_{\overline{x}} \tag{1}$$

Několik rad a myšlenek k měření

- Centrální limitní věta:
  Pro velký počet nezávislých proměnných se stále stejným rozdělením bude jejich rozdělení přibližně odpovídat rozdělení normálnímu, a to za předpokladu, že náhodné proměnné mají konečný rozptyl.

Několik rad a myšlenek k měření

- Máme normální rozdělení?
  - šikmost (skewness – třetí moment) γ_1 resp. g_1 pro vzorky
  - špičatost (kurtosis – čtvrtý moment) γ_2 resp. g_1 pro vzorky

$$g_1 = \frac{m_3}{m_2^{3/2}} = \frac{\frac{1}{n}\sum_{i=1} N(x_i - \bar{x})^3}{\left(\frac{1}{n}\sum_{i=1} N(x_i - \bar{x})^2\right)^{3/2}} \quad (1)$$

$$g_2 = \frac{m_4}{m_2^2} - 3 = \frac{\frac{1}{n}\sum_{i=1} N(x_i - \bar{x})^4}{\left(\frac{1}{n}\sum_{i=1} N(x_i - \bar{x})^2\right)^2} - 3 \quad (2)$$

  - $g_1 = 0$ normální
  - $g_1 < 0$ více vzorků je nalevo
  - $g_1 > 0$ naopak napravo
  - $g_2 = 0$ normální
  - $g_2 < 0$ těžké konce rozdělení
  - $g_2 > 0$ lehké konce

Několik rad a myšlenek k měření

- Skripta Fr. Šťastného
  `http://amper.ped.muni.cz/jenik/nejistoty/`

```
long promenna = 10000000L;
```

Paměťový model Javy

- *happens-before*
  - částečné uspořádání

The rules for *happens-before* are:

Program order rule. Each action in a thread *happens-before* every action in that thread that comes later in the program order.

Monitor lock rule. An unlock on a monitor lock *happens-before* every subsequent lock on that same monitor lock.[3]

Volatile variable rule. A write to a volatile field *happens-before* every subsequent read of that same field.[4]

Thread start rule. A call to `Thread.start` on a thread *happens-before* every action in the started thread.

Tabulka převzata z JCiP, Goetz

Paměťový model Javy

Thread termination rule. Any action in a thread *happens-before* any other thread detects that thread has terminated, either by successfully return from `Thread.join` or by `Thread.isAlive` returning `false`.

Interruption rule. A thread calling `interrupt` on another thread *happens-before* the interrupted thread detects the interrupt (either by having `InterruptedException` thrown, or invoking `isInterrupted` or `interrupted`).

Finalizer rule. The end of a constructor for an object *happens-before* the start of the finalizer for that object.

Transitivity. If *A happens-before B*, and *B happens-before C*, then *A happens-before C*.

Tabulka převzata z JCiP, Goetz

Paměťový model Javy

- Piggybacking
  - spojení happens-before pravidla s jiným pravidlem, obvykle monitorem nebo `volatile`
  - raději nepoužívat
  - příklad: **`http://kickjava.com/src/java/util/concurrent/FutureTask.java.htm`**
    - postaveno na `tryReleaseShared` happens-before `tryAcquireShared`
    - kombinace volatilní proměnné `runner`, do které `tryReleaseShared` zapisuje s program order

Paměťový model Javy

```
        void innerSet(V v) {
2       for (;;) {
        int s = getState();
4       if (ranOrCancelled(s))
            return;
6       if (compareAndSetState(s, RAN))
            break;
8       }
            result = v;
10          releaseShared(0);
            done();
12      }
```

```
        V innerGet() throws InterruptedException JavaDoc, ExecutionException Jav
2           acquireSharedInterruptibly(0);
            if (getState() == CANCELLED)
4               throw new CancellationException JavaDoc();
            if (exception != null)
6               throw new ExecutionException JavaDoc(exception);
            return result;
8       }
```

Paměťový model Javy

- líná inicializace

```
1  public class UnsafeLazyInitialization {
       private static Resource resource;
3
       public static Resource getInstance() {
5          if (resource == null)
               resource = new Resource(); // unsafe publication
7          return resource;
       }
9
       static class Resource {
11     }
   }
```

Paměťový model Javy

```
@ThreadSafe
public class SafeLazyInitialization {
    private static Resource resource;

    public synchronized static Resource getInstance() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }

    static class Resource {
    }
}
```

- líná inicializace thread-safe

Paměťový model Javy

```
@ThreadSafe
public class EagerInitialization {
    private static Resource resource = new Resource();

    public static Resource getResource() {
        return resource;
    }

    static class Resource {
    }
}
```

- „dychtivá“ inicializace
- využívá skutečnosti, že statické inicializátory jsou vždy dokončeny před použitím třídy

Paměťový model Javy

```
@ThreadSafe
public class ResourceFactory {
    private static class ResourceHolder {
        public static Resource resource = new Resource();
    }

    public static Resource getResource() {
        return ResourceFactory.ResourceHolder.resource;
    }

    static class Resource {
    }
}
```

- idiom líné inicializace s použitím holder class
- využívá líné inicializace tříd

Paměťový model Javy

```
public class DoubleCheckedLocking {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null) {
            synchronized (DoubleCheckedLocking.class) {
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }
```

- Double Checked Locking anti-pattern
- pomíjí možnost, že `resource` je v nedefinovaném stavu
- od Java 5.0 možno spravit použitím `volatile`
- nepoužívat
  - ani v C/C++!