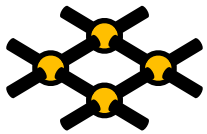
Vláknové programování
část IV

Lukáš Hejmánek, Petr Holub

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



Laboratoř pokročilých síťových technologií

PV192

2014–03–18

Přehled přednášky

ThreadFactory

- TPE vytváří vlákna pomocí ThreadFactory
  - metoda **`newThread`**
  - 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`**, nikoli od vlákna volajícího **`execute`**/**`submit`** (default)

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`**
  - využíváno metodou **`newSingleThreadExecutor`** (vrací omezený Executor – ačkoli implementace ve skutečnosti používá TPE s jediným vláknem)

Modifikace TPE

- Háčky pro modifikace
  - `beforeExecute`
  - `afterExecute`
  - `terminated`
- 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();
        }
    }
}
```

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`**

Java NIO

- Zavedeno v Javě 1.4 (JSR 51)
- Abstraktní třída **`Buffer`**
  - umožňuje držet pouze primitivní typy

```
ByteBuffer
CharBuffer
DoubleBuffer
FloatBuffer
IntBuffer
LongBuffer
ShortBuffer
```

- direct vs. non-direct buffery
  - přímé buffery se snaží vyhýbat zbytečným kopiiím mezi JVM a systémem
- vytváření pomocí metod
  - **`allocate`** – alokace požadované velikosti
  - **`allocateDirect`** – alokace požadované velikosti typu direct
  - **`wrap`** – zabalí existující pole bytů (bytearray)

Java NIO

- ByteBuffer
  - **`http://download.oracle.com/javase/6/docs/api/java/nio/ByteBuffer.html`**
  - přístup k binárním datům, např.

```
float  getFloat()
float  getFloat(int index)
 void  putFloat(float f)
 void  putFloat(int index, float f)
```

  - mapování souborů do paměti (**`FileChannel`**, metoda **`map`**)
  - čtění/vložení z/do bufferu bez parametru index (**`get`**/**`put`**) inkrementuje pozici
  - pokud není řečeno jinak, metody vrací odkaz na buffer – řetězení volání

```
buffer.putShort(10).putInt(0x00ABBCCD).putShort(11);
```

Java NIO

- Vlastnosti bufferů

  capacity celková kapacita bufferu

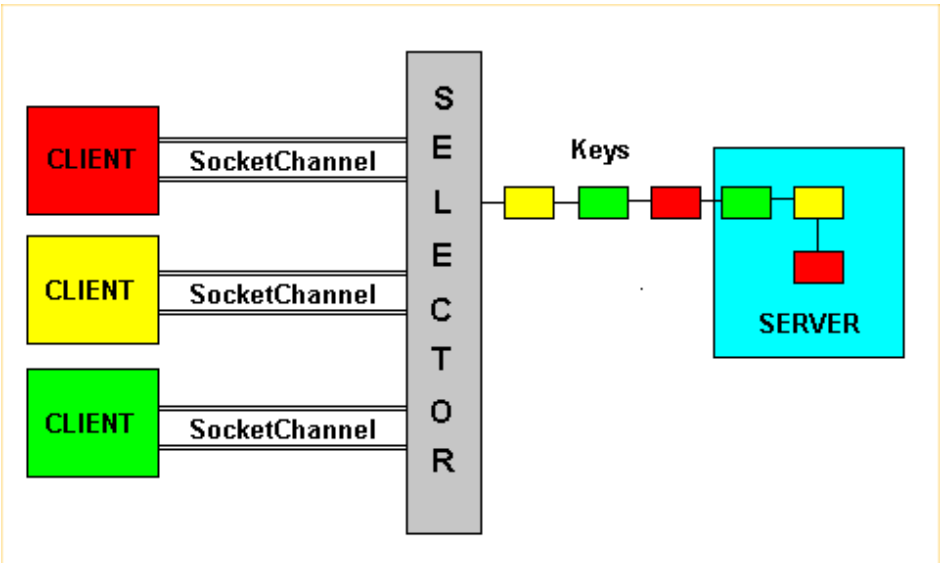
  limit umělý limit uvnitř bufferu, využití s metodami **`flip`** (nastaví limit na současnou pozici a skočí na pozici 0) či **`remaining`**

  mark pomocná značka, využití např. s metodou **`reset`** (skočí na označkovanou pozici)

```
buffer.position(10);
buffer.flip();
while (buffer.hasRemaining()) {
byte b = buffer.get();
// neco
}
```

Java NIO

- Selektor
  - serializace požadavků
  - výběr požadavků
- Klíč
  - identifikace konkrétního spojení



Zdroj: http://onjava.com/lpt/a/2672

Java NIO – Server

- Generický postup

```
create SocketChannel;
create Selector
associate the SocketChannel to the Selector
for(;;) {
  waiting events from the Selector;
  event arrived; create keys;
  for each key created by Selector {
    check the type of request;
    isAcceptable:
      get the client SocketChannel;
      associate that SocketChannel  to the Selector;
      record it for read/write operations
      continue;
   isReadable:
      get the client SocketChannel;
      read from the socket;
      continue;
   isWriteable:
      get the client SocketChannel;
      write on the socket;
      continue;
  }
}
```

Zdroj: http://onjava.com/lpt/a/2672

Java NIO – Server

```
// Create the server socket channel
ServerSocketChannel server = ServerSocketChannel.open();
// nonblocking I/O
server.configureBlocking(false);
// host-port 8000
server.socket().bind(new java.net.InetSocketAddress(host,8000));
// Create the selector
Selector selector = Selector.open();
// Recording server to selector (type OP_ACCEPT)
server.register(selector,SelectionKey.OP_ACCEPT);
```

Zdroj: http://onjava.com/lpt/a/2672

Java NIO – Server

```
// Infinite server loop
for(;;) {
  // Waiting for events
  selector.select();
  // Get keys
  Set keys = selector.selectedKeys();
  Iterator i = keys.iterator();

  // For each keys...
  while(i.hasNext()) {
    SelectionKey key = (SelectionKey) i.next();

    // Remove the current key
    i.remove();

    // if isAccetable = true
    // then a client required a connection
    if (key.isAcceptable()) {
      // get client socket channel
      SocketChannel client = server.accept();
      // Non Blocking I/O
      client.configureBlocking(false);
      // recording to the selector (reading)
      client.register(selector, SelectionKey.OP_READ);
      continue;
    }
```

Zdroj: http://onjava.com/lpt/a/2672

Java NIO – Server

```
    // if isReadable = true
2   // then the server is ready to read
    if (key.isReadable()) {
4
      SocketChannel client = (SocketChannel) key.channel();
6
      // Read byte coming from the client
8     int BUFFER_SIZE = 32;
      ByteBuffer buffer = ByteBuffer.allocate(BUFFER_SIZE);
10    try {
        client.read(buffer);
12    }
      catch (Exception e) {
14      // client is no longer active
        e.printStackTrace();
16      continue;
      }
18
      // Show bytes on the console
20    buffer.flip();
      Charset charset=Charset.forName(''ISO-8859-1'');
22    CharsetDecoder decoder = charset.newDecoder();
      CharBuffer charBuffer = decoder.decode(buffer);
24    System.out.print(charBuffer.toString());
      continue;
26  }
  }
28}
```

Java NIO

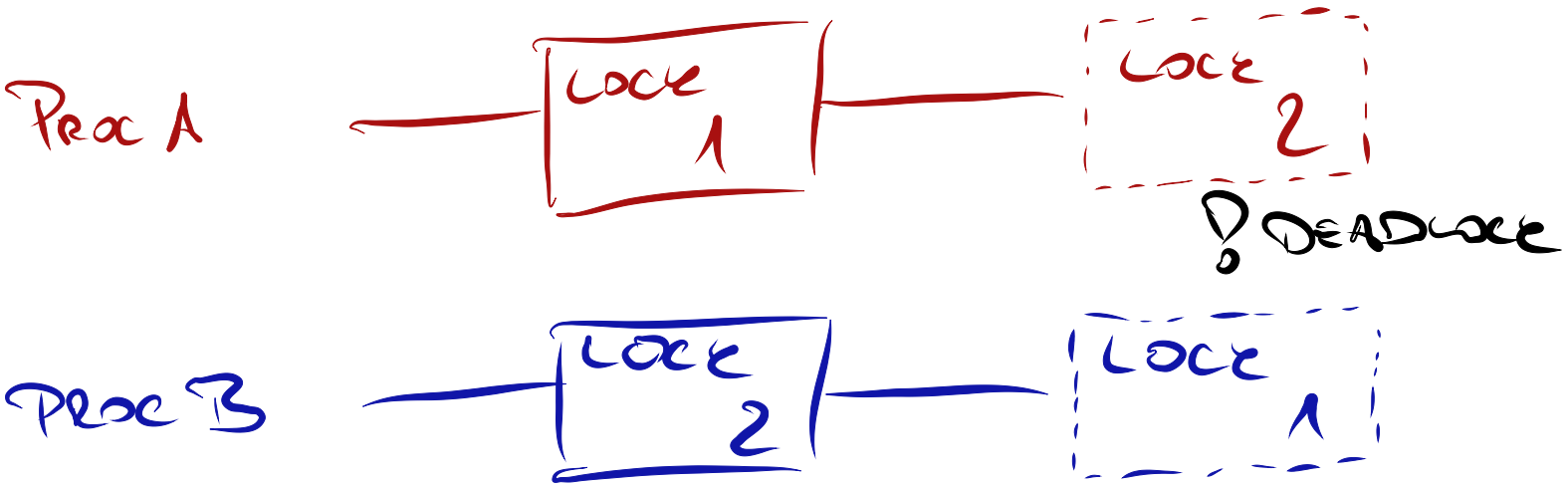
- Další čtení:
  - **http://onjava.com/lpt/a/2672**
  - **http://onjava.com/lpt/a/5127**
  - **http://download.oracle.com/javase/6/docs/api/java/nio/channels/Selector.html**
  - **http://download.oracle.com/javase/6/docs/api/java/nio/channels/SelectionKey.html**

Asynchronní programování versus vlákna

- Asynchronní programování
  - \+ umožňuje obsluhovat řádově větší množství klientů
  - – za cenu zvýšení latence
  - – složitější, náchylnější na chyby
- Vláknové programování
  - \+ jednodušší
  - \+ poměrně efektivní do „rozumného” počtu vláken
  - – nativní vlákna nejsou stavěna na (deseti)tisíce vláken a více
- Potenciálně lze kombinovat

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) {
2           if (fromAcct.lock.tryLock()) {
                try {
4                   if (toAcct.lock.tryLock()) {
                        try {
6                           if (fromAcct.getBalance().compareTo(amount) < 0)
                                throw new InsufficientFundsException();
8                           else {
                                fromAcct.debit(amount);
10                              toAcct.credit(amount);
                                return true;
12                          }
                        } finally {
14                          toAcct.lock.unlock();
                        }
16                  }
                } finally {
18                  fromAcct.lock.unlock();
                }
20          }
            if (System.nanoTime() < stopTime)
22              return false;
            NANOSECONDS.sleep(fixedDelay + rnd.nextLong() % randMod);
24      }
```

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 {
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 synchronized Image getImage() {
            Image image = new Image();
16          for (Taxi t : taxis)
                image.drawMarker(t.getLocation());
18          return image;
        }
20  }
```

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      }
    }
```

```
    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 synchronized Image getImage() {
            Image image = new Image();
16          for (Taxi t : taxis)
                image.drawMarker(t.getLocation());
18          return image;
        }
20  }
```

- **`setLocation` → `notifyAvailable`**
- **`getImage` → `getLocation`**

Otevřená volání

- Otevřené volání (open call)
  - volání metody, kdy volající nedrží žádný zámek
  - 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 {
2       @GuardedBy("this") private Point location, destination;
        private final Dispatcher dispatcher;
4
        public Taxi(Dispatcher dispatcher) { this.dispatcher = dispatcher; }
6
        public synchronized Point getLocation() { return location; }
8
        public void setLocation(Point location) {
10          boolean reachedDestination;
            synchronized (this) {
12              this.location = location;
                reachedDestination = location.equals(destination);
14          }
            if (reachedDestination)
16              dispatcher.notifyAvailable(this);
        }
18
        public synchronized Point getDestination() { return destination; }
20
        public synchronized void setDestination(Point destination) {
22          this.destination = destination;
        }
24  }
```

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 Image getImage() {
        Set<Taxi> copy;
        synchronized (this) {
            copy = new HashSet<Taxi>(taxis);
        }
        Image image = new Image();
        for (Taxi t : copy)
            image.drawMarker(t.getLocation());
        return image;
    }
}
```

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

```
        t.setPriority(Thread.MIN_PRIORITY); // 1
2       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 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

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

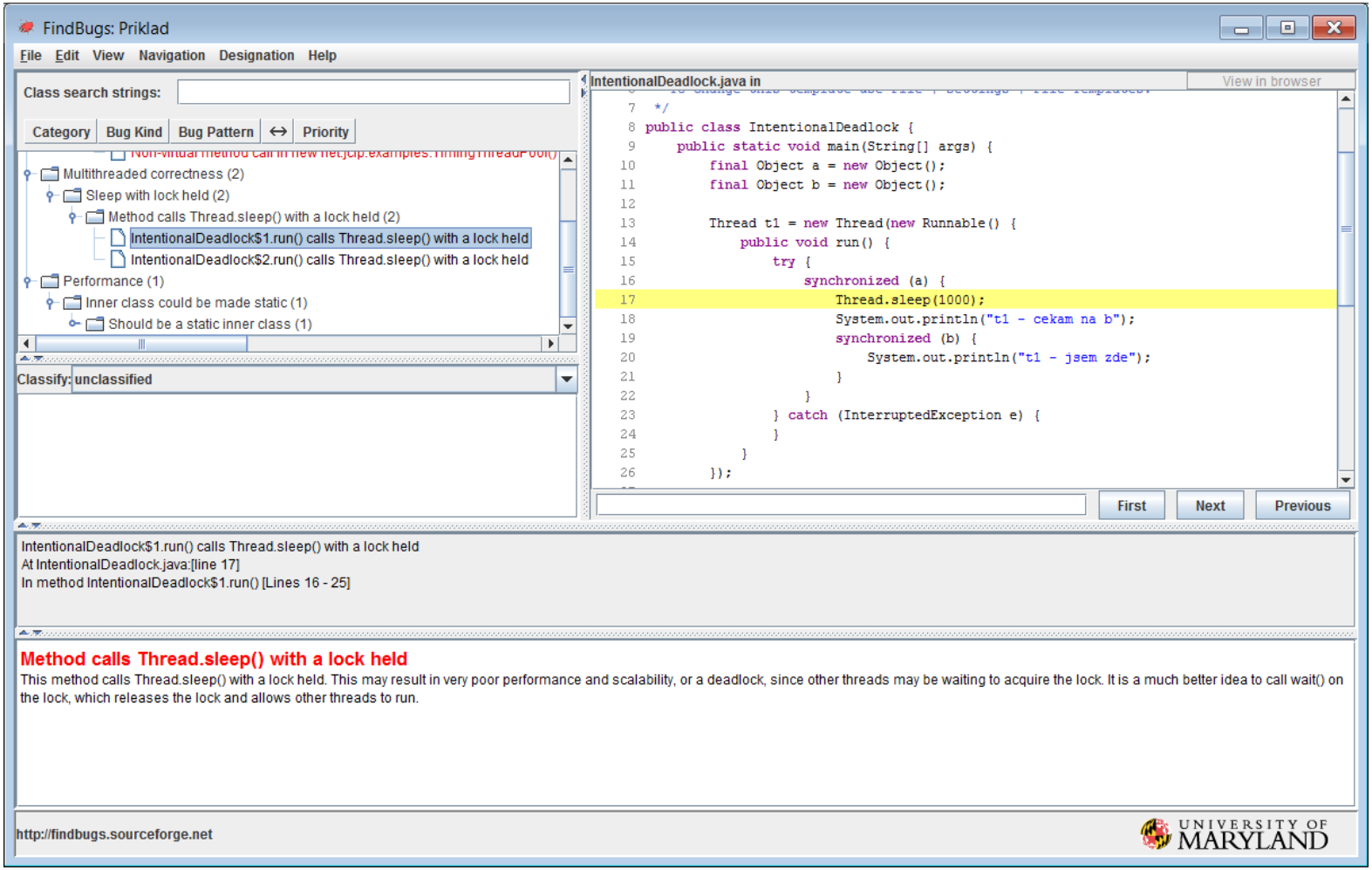
```
Found one Java-level deadlock:
============================
"Thread-1":
  waiting to lock monitor 0x020d53ac (object 0x243e6928, a java.lang.Object),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x020d6c74 (object 0x243e6930, a java.lang.Object),
  which is held by "Thread-1"

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 – lokální objekt, který není nikdy publikován na haldu a je tudíž thread-local)
  - 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 – lokální objekt, který není nikdy publikován na haldu a je tudíž thread-local)
  - 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

Domácí úloha

1. Naimplementuje echo server, který bude na zadaných portech (tcp:portnum1, udp:portnum2,…) poslouchat a implementovat echo, tedy vypisovat vstup přes daný port přijatý zpět. Program se ukončí na Q+Enter do terminálu.
2. Termín: 6. 4. 2014

Domácí úloha

1. Naimplementuje webový harvestor, který rekurzivně stahuje a hledá v nich zadaný text.
   - Soubory není třeba ukládat.
   - Všechny řádky staženého souboru spojte do jednoho, najděte všechny výskyty `<a href="...">` a uložte si relativní i absolutní URL do fronty k dalšímu stažení.
   - Pokud najdete zadaný text, vytiskněte daný řádek do GUI okna spolu s URL, kde byl nalezen.
   - Program bude mít GUI zobrazující hlavičky aktuálně stahovaných dokumentů.
   - Harvestor skončí pokud narazí na max. limit stránek (konfigurovatelný), nebo nebude mít další práci.
   - Harvestor bude respektovat limit (konfigurovatelný) na počet připojení na daný HTTP server.
2. Pro implementaci stahování vytvořte vlastní thread pool (rozšířením třídy ThreadPool), který se bude dynamický zvětšovat/zmenšovat podle počtu čekajících požadavků ve frontě. Tj. začne růst nad corePoolSize and již před zaplněním fronty – po dosažení např. 75% zaplnění fronty.
3. Termín: 20. 4. 2014