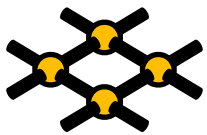
Vláknové programování
část III

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



Laboratoř pokročilých síťových technologií

PV192
2010–03–11

Přehled přednášky

Vlákna v jazyce Java

Viditelnost a synchronizace

Ukončování vláken

Monitory a synchronizace

Signalizace a suspend

Vlákna v jazyce Java

- Mechanismy
  - potomek třídy Thread
  - Executory
  - objekt implementující vlákno pro Executor
    - objekt implementující interface Runnable (metoda **`run()`**)
    - objekt implementující interface Callable (metoda **`call()`**)
- Synchronizace a viditelnost operací
- Implementace monitorů pomocí **`synchronized`**
- Signalizace mezi objekty: **`wait`**, **`notify`**, **`notifyAll`**
- Knihovny **`java.util.concurrent`**

Třída Thread

- Základní třída pro vlákna
- Metoda **`run()`**
  - „vnitřnosti“ vlákna
  - přepisuje se vlastním kódem
- Metoda **`start()`**
  - startování vlákna
  - za normálních okolností nepřepisuje!
  - pokud už se přepisuje, je třeba volat **`super.start()`**

Třída Thread

```
1  public class PrikladVlakna {
2
3      static class MojeVlakno extends Thread {
4          MojeVlakno(String jmenoVlakna) {
5              super(jmenoVlakna);
6          }
7
8          public void run() {
9              for (int i = 0; i < 10; i++) {
10                 System.out.println(this.getName() +
11                         ": pocitam vzbuzeni - " + (i + 1));
12                 try {
13                     sleep(Math.round(Math.random()));
14                 } catch (InterruptedException e) {
15                     System.out.println(this.getName() +
16                             ": probudil jsem se nenadale! :-|");
17                 }
18             }
19         }
20     }
21
22     public static void main(String[] args) {
23         new MojeVlakno("vlakno1").start();
24         new MojeVlakno("vlakno2").start();
25     }
}
```

Viditelnost a synchronizace operací

(nic) > `volatile` > `AtomicXXX` > `synchronized`, explicitní zámky

- problém viditelnosti změn
- problém atomicity operací
  - např. přiřazení do 64-bitového typu (**`long`**, **`double`**) není nutně atomický!
- problém synchronizace při změnách hodnot

Viditelnost a synchronizace operací

(nic) > **volatile** > **AtomicXXX** > **synchronized**, explicitní zámky

```
public class Nic {
    private static long cislo = 10000000L;
    private static boolean pripraven = false;

    public static class Vlakno extends Thread {
        public void run() {
            while (!pripraven) {
                Thread.yield();
            }
            System.out.println(cislo);
        }
    }

    public static void main(String[] args) {
        new Vlakno().start();
        cislo = 42L;
        pripraven = true;
    }
}
```

Viditelnost a synchronizace operací

(nic) > **`volatile`** > **`AtomicXXX`** > **`synchronized`**, explicitní zámky

- Jak to dopadne?
  - neatomičnost 64-bitového přiřazení
  - přeuspořádání přiřazení
  - kterákoli ze změn hodnot nemusí být viditelná
  - *jakkoli…*
    - 10.000.000 nebo 42 nebo něco jiného (neatomičnost – vlákno se může trefit mezi přiřazení horní a dolní poloviny 64 b operace)
    - ale také může navždy cyklit (Vlakno neuvidí nastavení pripraven)

pročež platí

1. Pokud více vláken čte jednu proměnnou, musí se řešit viditelnost.
2. Pokud více vláken zapisuje do jedné proměnné, musí se synchronizovat.

Viditelnost a synchronizace operací

(nic) > **`volatile`** > **`AtomicXXX`** > **`synchronized`**, explicitní zámky

- **`volatile`**
  - zajišťuje viditelnost změn mezi vlákny
  - překladač nesmí dělat presumpce/optimalizace, které by mohly ovlivnit viditelnost
  - u 64 b přiřazení zajišťuje atomičnost
  - nezajišťuje atomičnost operace načti–změň–zapiš!
    - nelze použít pro thread-safe **`i++`**
  - lze použít pokud jsou splněny obě následující podmínky
    1. nová hodnota proměnné nezávisí na její předchozí hodnotě
    2. proměnná se nevyskytuje v invariantech jiných proměnných

    - např. příznak ukončení nebo jiné události, který nastavuje pouze jedno vlákno – pomohlo by v příkladě třídy Nic (slajd 7)
    - příklady použití:
      `http://www.ibm.com/developerworks/java/library/j-jtp06197.html`
  - pokud si nejsme jisti, použijeme raději silnější synchronizaci

Viditelnost a synchronizace operací

(nic) > **volatile** > **AtomicXXX** > **synchronized**, explicitní zámky

```
1  public class VolatileInvariant {
       private volatile int horniMez, dolniMez;
3
       public int getHorniMez() { return horniMez; }
5
       public void setHorniMez(int horniMez) {
7          if (horniMez < this.dolniMez)
               throw new IllegalArgumentException("Horni < dolni!");
9          else
               this.horniMez = horniMez;
11     }

13     public int getDolniMez() { return dolniMez; }

15     public void setDolniMez(int dolniMez) {
           if (dolniMez > this.horniMez)
17             throw new IllegalArgumentException("Dolni > horni!");
           else
19             this.dolniMez = dolniMez;
       }
21 }
```

Viditelnost a synchronizace operací

(nic) > **`volatile`** > **`AtomicXXX`** > **`synchronized`**, explicitní zámky

- **`AtomicXXX`**
  - zajišťuje viditelnost
  - zajišťuje atomičnost operace načti–změň–zapiš nad objektem
    - potřebujeme-li udělat více takových operací synchronně, nelze použít
- **`synchronized`**, explicitní zámky
  - zajištění viditelnosti
  - zajištění vyloučení (a tedy i atomičnosti) v kritické sekci

Publikování a únik

- Publikování objektu
  - zveřejnění odkazu na objekt
  - thread-safe třídy nesmí publikovat objekty bez zajištěné synchronizace
  - nepřímé publikování
    - publikování jiného objektu, přes něhož je daný objekt *dosažitelný*
    - např. publikování kolekce obsahující objekt
    - např. instance vnitřní třídy obsahuje odkaz na vnější třídu
  - „vetřelecké“ (*alien*) metody
    - metody, jejichž chování není plně definováno samotnou třídou
    - všechny metody, které nejsou **`private`** nebo **`final`** – mohou být přepsány v potomkovi
    - předání objektu vetřelecké metodě = publikování objektu

Publikování a únik

- Únik stavu objektu
  - publikování reference na interní měnitelné (*mutable*) objekty
  - v níže uvedeném příkladě může klient měnit pole **`states`**
  - potřeba hlubokého kopírování (*deep copy*)

```
1  import java.util.Arrays;

3  public class UnikStavu {
       private String[] stavy = new String[]{"Stav1", "Stav2", "Stav3"};
5
       public String[] getStavySpatne() {
7          return stavy;
       }
9
       public String[] getStavySpravne() {
11         return Arrays.copyOf(stavy, stavy.length);
       }
13 }
```

Publikování a únik

- Únik z konstruktoru
  - až do návratu z konstruktoru je objekt v „rozpracovaném“ stavu
  - publikování objektu v tomto stavu je obzvláště nebezpečné
  - pozor na skryté publikování přes **`this`** v rámci instance vnitřní třídy
    - registrace listenerů na události

```
1  public class UnikZKonstruktoru {
       public UnikZKonstruktoru(EventSource zdroj) {
3          zdroj.registerListener(
                   new EventListener() {
5                      public void onEvent(Event e) {
                           zpracujUdalost(e);
7                      }
                   }
9          );
       }
11 }
```

Publikování a únik

- Únik z konstruktoru
  - když musíš, tak musíš… ale aspoň takto:
    1. vytvořit soukromý konstruktor
    2. vytvořit veřejnou factory

```
1  public class BezpecnyListener {
       private final EventListener listener;
3
       private BezpecnyListener() {
5          listener = new EventListener() {
               public void onEvent(Event e) {
7                  zpracujUdalost(e);
               }
9          };
       }
11
       public static BezpecnyListener novaInstance(EventSource zdroj) {
13         BezpecnyListener bl = new BezpecnyListener();
           zdroj.registerListener(bl.listener);
15         return bl;
       }
17 }
```

Thead-safe data

- *ad hoc*
  - zodpovědnost čistě ponechaná na implmentaci
  - nepoužívá podporu jazyka
  - pokud možno nepoužívat

Thead-safe data

- data omezená na zásobník
  - data na zásobníku patří pouze danému vláknu
  - týká se lokálních proměnných
    - u primitivních lokálních proměnných nelze získat ukazatel a tudíž je nelze publikovat mimo vlákno
    - ukazatele na objekty je třeba hlídat (programátor), že se objekt nepublikuje a zůstává lokální
    - lze používat ne-thread-safe objekty, ale je rozumné to dokumentovat (pro následné udržovatele kódu)

```
1  import java.util.Collection;

3  public class PocitejKulicky {
       public class Kulicka {
5      }

7      public int pocetKulicek(Collection<Kulicka> kulicky) {
           int pocet = 0;
9          for (Kulicka kulicka : kulicky) {
               pocet++;
11         }
           return pocet;
13     }
   }
```

Thead-safe data

- **`ThreadLocal`**
  - data asociovaná s každým vláknem zvlášť, ukládá se do Thread
  - používá se často v kombinaci se Singletony a globálními proměnnými
  - JDBC spojení na databázi nemusí být thread-safe

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class PrikladTL {
    private static ThreadLocal<Connection> connectionHolder =
            new ThreadLocal<Connection>() {
                protected Connection initialValue() {
                    try {
                        return DriverManager.getConnection("DB_URL");
                    } catch (SQLException e) {
                        return null;
                    }
                }
            };

    public static Connection getConnection() {
        return connectionHolder.get();
    }
}
```

Thead-safe data

- Neměnné (*immutable*) objekty
  - neměnný objekt je takový
    - jehož stav se nemůže změnit, jakmile je zkonstruován
    - všechny jeho pole jsou **`final`**
    - je řádně zkonstruován (nedojde k úniku z konstruktoru)
  - neměnné objekty jsou automaticky thread-safe
  - pokud potřebujeme provést složenou akci atomicky, můžeme ji zabalit do vytvoření neměnného objektu na zásobníku a jeho publikaci pomocí **`volatile`** odkazu
    - nemůžeme předpokládat atomičnost načti–změň–zapiš (**`i++`** chování)
  - díky levné alokaci[1] nových objektů (JDK verze 5 a výš) se dají efektivně používat

[1] Do JDK 5.0 se používalo **`ThreadLocal`** pro recyklaci bufferů metody **`Integer.toString`**. Od verze 5.0 se vždy alokuje nový buffer, je to rychlejší.

Thead-safe data

- Neměnné (*immutable*) objekty

```
public class NemennaCache {
    private final String lastURL;
    private final String lastContent;

    public NemennaCache(String lastURL, String lastContent) {
        this.lastURL = lastURL;
        this.lastContent = lastContent;
    }

    public String vemZCache(String url) {
        if (lastURL == null || !lastURL.equals(url))
            return null;
        else
            return lastContent;
    }
}
```

Thead-safe data

- Neměnné (*immutable*) objekty

```
public class PouzitiCache {
    private volatile NemennaCache cache = new NemennaCache(null, null);

    public String nactiURL(String URL) {
        String obsah = cache.vemZCache(URL);
        if (obsah == null) {
            obsah = fetch(URL);
            cache = new NemennaCache(URL, obsah);
            return obsah;
        } else
            return obsah;
    }
}
```

Bezpečné publikování

- Nebezpečné publikování
  - publikování potenciálně nedokončeného měnitelného objektu
  - takto by šlo publikovat pouze neměnné objekty (lépe s použitím **`volatile`**)

```
1  public class NebezpecnePublikovani {
       public class Pytlicek {
3          public int hodnota;

5          public Pytlicek(int hodnota) {
               this.hodnota = hodnota;
7          }
       }
9
       public Pytlicek pytlicek;
11
       public void incializujPytlicek(int i) {
13         pytlicek = new Pytlicek(i);
       }
15 }
```

Bezpečné publikování

- Způsoby bezpečného publikování měnitelných objektů
  1. inicializace odkazu ze statického inicializátoru
  2. uložení odkazu do **`volatile`** nebo **`AtomicReference`** pole
  3. uložení odkazu od **`final`** pole – bezpečné až po návratu z konstruktoru
  4. uložení odkazu do pole, které je chráněno zámky/monitorem
  5. uložení do thread-safe kolekce (**`Hashtable`**, **`synchronizedMap`**, **`ConcurrentMap`**, **`Vector`**, **`CopyOnWriteArray{List,Set}`**, **`synchronized{List,Set}`**, **`BlockingQueue`**, **`ConcurrentLinkedQueue`**)
     - objekt i odkaz musí být publikovány současně

Bezpečné publikování

- Efektivně neměnné objekty
  - pokud se k objektu chováme jako neměnnému
  - bezpečné publikování je dostatečné
- Měnitelné objetky
  - bezpečné publikování zajistí pouze viditelnost ve výchozím stavu
  - změny je třeba synchronizovat (zámky/monitory)

Bezpečné sdílení objektů

- Uzavřené ve vlákně
- Sdílené jen pro čtení
  - neměnné a efektivně neměnné objekty
- Thread-safe objekty
  - zajišťují si synchronizaci uvnitř samy
- Chráněné objekty
  - zabalené do thread-safe konstrukcí (thread-safe objektů, chráněny zámkem/monitorem)

Ukončování vláken

- Vlákna by se měla zásadně ukončovat dobrovolně a samostatně
  - metoda **`Thread.stop()`** je deprecated
  - násilné ukončení vlákna může nechat systém v nekonzistentním stavu
    - výjimka **`ThreadDeath`** tiše odemkne všechny monitory, které vlákno drželo
    - objekty, které byly monitory chráněny, mohou být v nekonzistentním stavu
  - **`http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html`**
- Jak na to?
  - zavést si proměnnou, která bude signalizovat požadavek na ukončení nebo
  - využít příznak **`isInterrupted()`**
  - použití metody **`interrupt()`**
  - použití I/O blokujících omezenou dobu
- Vlákna lze násilně ukončovat ve speciálních případech
  - ExecutorService
  - Futures

Ukončování vláken

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

3  public class PrikladUkonceni {
       static class MojeVlakno extends Thread {
5          private volatile boolean ukonciSe = false;

7          public void run() {
               while (!ukonciSe) {
9                  try {
                       System.out.println("...chrnim...");
11                     sleep(1000);
                   } catch (InterruptedException e) {
13                     System.out.println("Vzbudil jsem se necekane!");
                   }
15             }
           }
17
           public void skonci() {
19             ukonciSe = true;
           }
21     }
```

Ukončování vláken

```
    public static void main(String[] args) {
24      try {
            MojeVlakno vlakno = new MojeVlakno();
26          vlakno.start();
            sleep(2000);
28          vlakno.skonci();
            vlakno.interrupt();
30          vlakno.join();
        } catch (InterruptedException e) {
32          e.printStackTrace();
        }
34  }
}
```

`synchronized` a monitory

- Monitory – Hoare, Dijkstra, Hansen, cca 1974
  - vynucuje serializovaný přístup k objektu
  - implementace Hoarových monitorů pro Javu:
    `http://www.engr.mun.ca/%7Etheo/Misc/monitors/monitors.html`
    `http://www.javaworld.com/javaworld/jw-10-2007/jw-10-monitors.html`
- `synchronized` – základní nástroj pro vyloučení v kritické sekci
  - v Javě se označuje jako monitor
  - synchronizuje se na explicitně uvedeném objektu (raději `final`) nebo (implicitně) na `this`

`synchronized` a monitory

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

`synchronized` a monitory

- *Java monitor pattern*

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

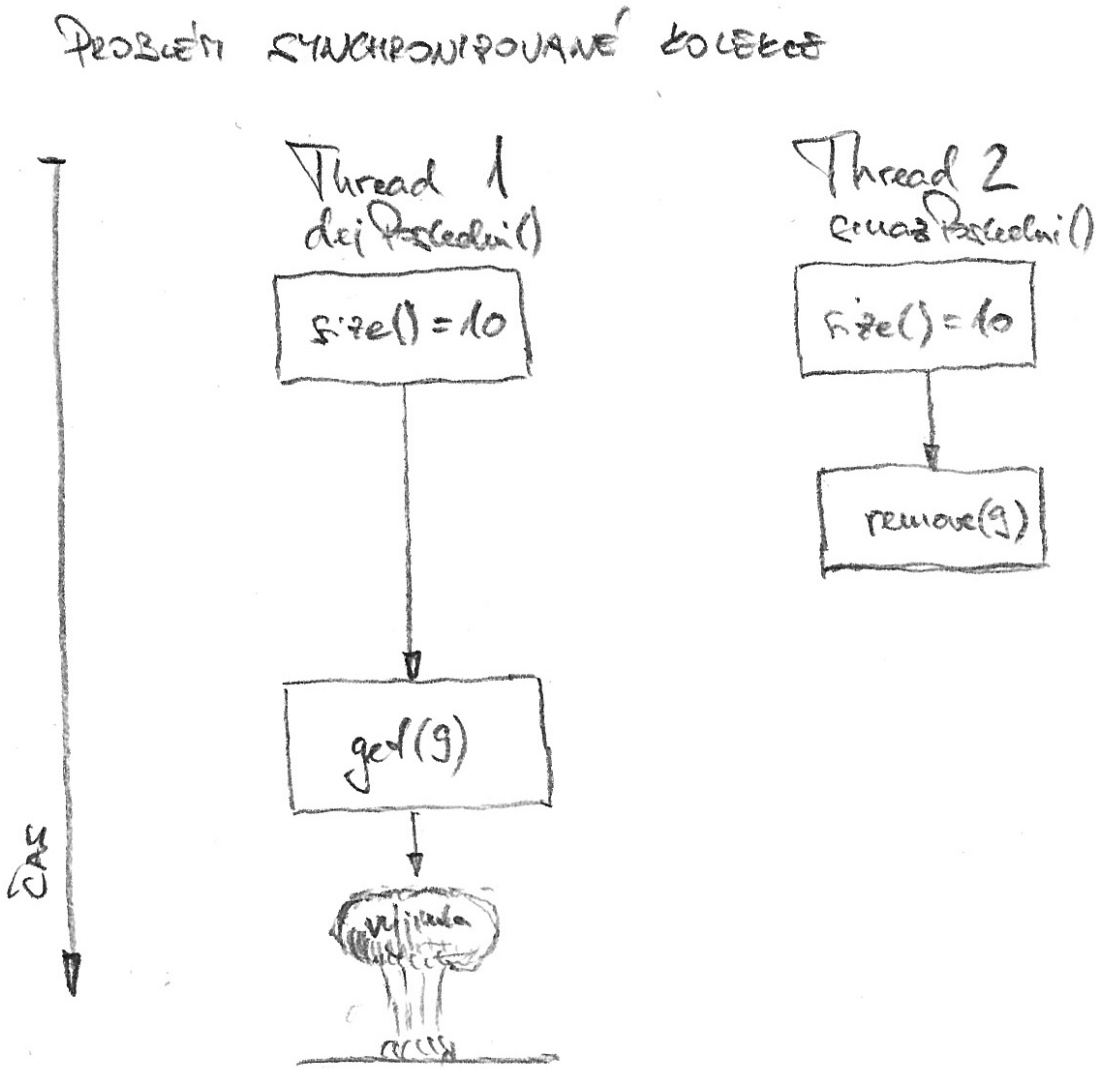
Sychronized Collections

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

Sychronized Collections

```
1  import java.util.Vector;
2
3  public class PodlaKolekce {
4      public static Object dejPosledni(Vector v) {
5          int posledni = v.size() - 1;
6          return v.get(posledni);
7      }
8
9      public static void smazPosledni(Vector v) {
10         int posledni = v.size() - 1;
11         v.remove(posledni);
12     }
13 }
```

Sychronized Collections



Sychronized Collections

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

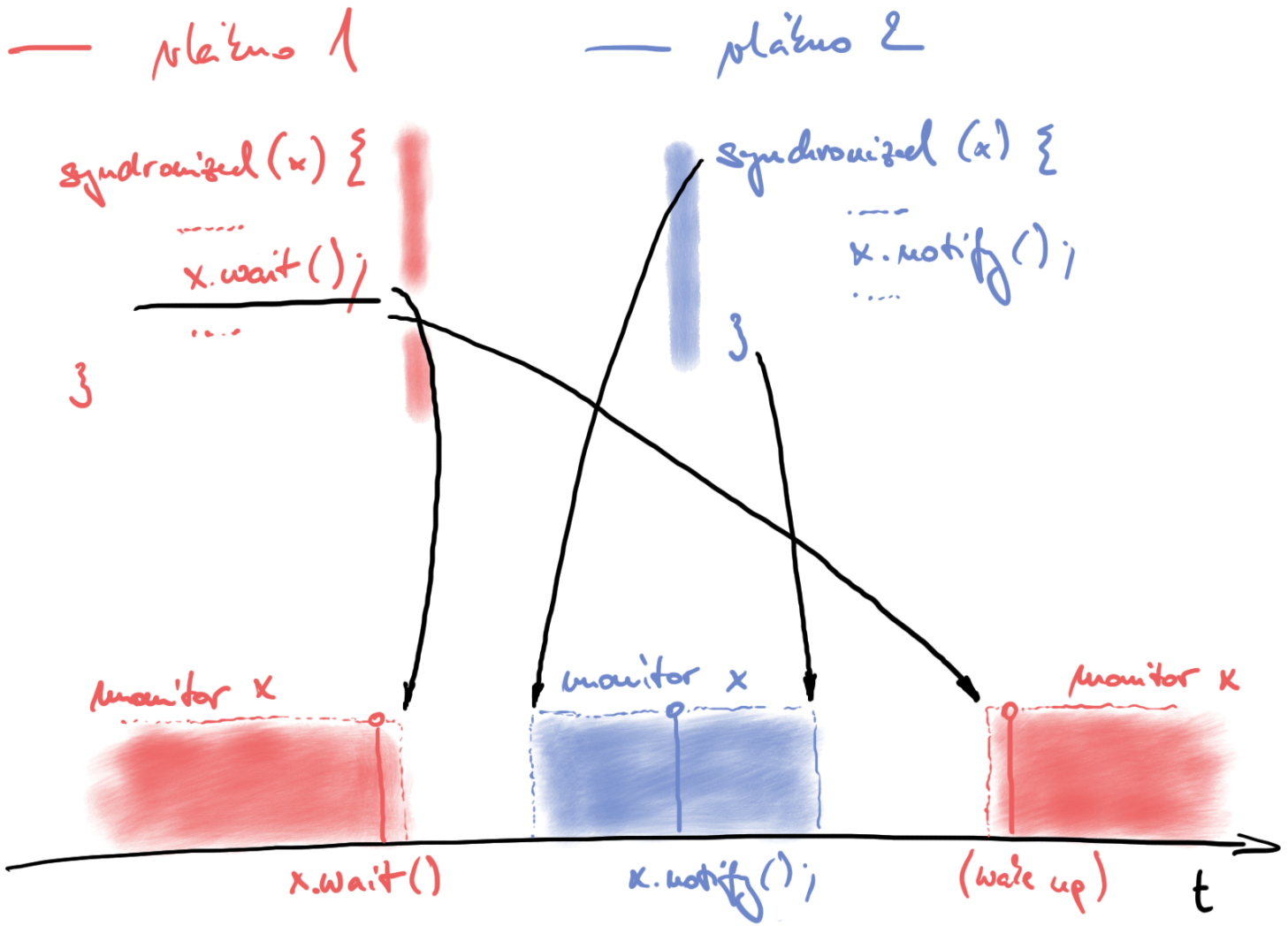
Sychronized Collections

```
1  import java.util.Vector;
2
3  public class RucneSyncnutaKolekce {
4      public static Object dejPosledni(Vector v) {
5          synchronized (v) {
6              int posledni = v.size() - 1;
7              return v.get(posledni);
8          }
9      }
10
11     public static void smazPosledni(Vector v) {
12         synchronized (v) {
13             int posledni = v.size() - 1;
14             v.remove(posledni);
15         }
16     }
17
   }
```

Signalizace mezi objekty

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

Signalizace mezi objekty



Suspendování vláken

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

Suspendování vláken

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

public class PrikladSuspendu {
    static class MojeVlakno extends Thread {
        private volatile boolean ukonciSe = false;
        private volatile boolean spi = false;

        public void run() {
            while (!ukonciSe) {
                System.out.println("...makam...");
                try {
                    sleep(500);
                    synchronized (this) {
                        while (spi) {
                            wait();
                        }
                    }
                } catch (InterruptedException e) {
                    System.out.println("Necekane probuzeni!");
                }
            }
            System.out.println("...domakal jsem...");
        }

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

Suspendování vláken

```
        public void usni() {
30          spi = true;
        }
32
        public void vzbudSe() {
34          spi = false;
            synchronized (this) {
36              this.notify();
            }
38      }
    }
40
    public static void main(String[] args) {
42      try {
            MojeVlakno vlakno = new MojeVlakno();
44          vlakno.start();
            sleep(2000);
46          vlakno.usni();
            sleep(2000);
48          vlakno.vzbudSe();
            sleep(2000);
50          vlakno.skonci();
            vlakno.join();
52      } catch (InterruptedException e) {
            e.printStackTrace();
54      }
    }
56 }
```

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

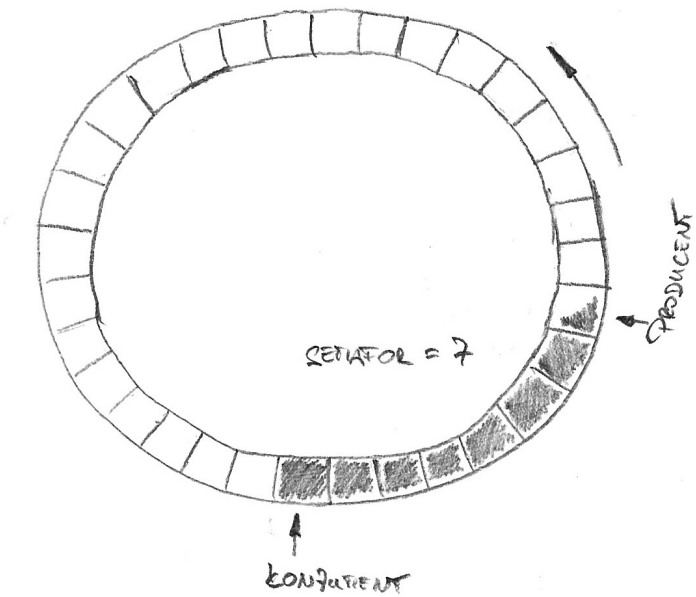
- Deque a BlockingDeque
  - umožňují vybírat prvky ze začátku i z konce fronty
  - normální konzumenti vybírají prvky ze začátku fronty
  - vlákna, která se „nudí“ mohou převzít práci z konce fronty
  - např. udržování fronty per vlákno, „nudící se“ vlákna mohou koukat do cizích front
  - vhodné např.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